Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars

Eric Badouel, Loïc Hélouët, Georges-Edouard Kouamou, Christophe Morvan, Robert Fondze Jr Nsaibirni

Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars

Eric Badouel
Inria and LIRIMA
Campus de Beaulieu
35042 Rennes, France
eric.badouel@inria.fr

Loïc Hélouët
Inria
Campus de Beaulieu
35042 Rennes, France
loic.helouet@inria.fr

Georges-Edouard Kouamou
ENSP and LIRIMA
BP 8390, Yaoundé, Cameroon
georges.kouamou@lirima.org

Christophe Morvan
Université Paris-Est
UPEMLV, F-77454
Marne-la-Vallée, France
christophe.morvan@u-pem.fr

Robert Fondze Jr Nsaibirni
UY1 and LIRIMA
BP 812, Yaoundé, Cameroon
nsairobby@gmail.com

ABSTRACT

This paper presents a purely declarative approach to artifact-centric collaborative systems, a model which we introduce in two stages. First, we assume that the workspace of a user is given by a mindmap, shortened to a map, which is a tree used to visualize and organize tasks in which he or she is involved, with the information used for the resolution of these tasks. We introduce a model of *guarded attribute grammar*, or GAG, to help the automation of updating such a map. A GAG consists of an underlying grammar, that specifies the logical structure of the map, with semantic rules which are used both to govern the evolution of the tree structure (how an open node may be refined to a subtree) and to compute the value of some of its attributes (which derives from contextual information). The map enriched with this extra information is termed an *active workspace*. Second, we define collaborative systems by making the various user's active workspaces communicate with each other. The communication uses message passing without shared memory thus enabling convenient distribution on an asynchronous architecture. We present some formal properties of the model of guarded attribute grammars, then a language for their specification and we illustrate the approach on a case study for a disease surveillance system.

Keywords

Business Artifacts, Case Management, Attribute Grammars

1. INTRODUCTION

This paper presents a modular and purely declarative model of artifact-centric collaborative systems, which is user-centric, easily distributable on an asynchronous architecture and reconfigurable without resorting to global synchronizations.

Traditional Case Management Systems rely on workflow models. The emphasis is put on the orchestration of activities involving humans (the *stakeholders*) and software systems, in order to achieve some global objective. In this context, stress is often put on control and coordination of tasks required for the realization of a particular service. Such systems are usually modeled with centralized and state-based formalisms like automata, Petri nets or statecharts. They can also directly be specified with dedicated notations like BPEL [33] or BPMN [22].

One drawback of existing workflow formalisms is that *data* exchanged during the processing of a task play a secondary role when not simply ignored. However, data can be tightly connected with control flows and should not be overlooked. Actually, data contained in a request may influence its processing. Conversely different decisions during the treatment of a case may produce distinct output-values.

Similarly, stakeholders are frequently considered as second class citizens in workflow systems: They are modeled as plain resources, performing specific tasks for a particular case, like machines in assembly lines. As a result, workflow systems are ideal to model fixed production schemes in manufactures or organizations, but can be too rigid to model open architectures where the evolving rules and data require more flexibility.

Data-centric workflow systems were proposed by IBM [32, 24, 10]. They put stress on the exchanged documents, the so-called *Business Artifacts*, also known as *business entities with lifecycles*. An artifact is a document that conveys all the information concerning a particular case from its inception in the system until its completion. It contains all the relevant information about the entity together with a lifecycle that models its possible evolutions through the business process. Several variants presenting the life cycle of an artifact by an automaton, a Petri net [29], or logical formulas depicting legal successors of a state [10] have been proposed. However, even these variants remain state-based centralized models in which stakeholders do not play a central role.

Guard-Stage-Milestone (GSM), a declarative model of the lifecycle of artifacts was recently introduced in [25, 11]. This

model defines *Guards*, *Stages* and *Milestones* to control the enabling, enactment and completion of (possibly hierarchical) activities. The GSM lifecycle meta-model has been adopted as a basis of the OMG standard *Case Management Model and Notation* (CMMN). The GSM model allows for dynamic creation of subtasks (the *stages*), and handles data attributes. Furthermore, guards and milestones attached to stages provide declarative descriptions of tasks inception and termination. However, interaction with users are modeled as incoming messages from the environment, or as events from low-level (atomic) stages. In this way, users do not contribute to the choice of a workflow for a process. The semantics of GSM models is given in terms of global snapshots. Events can be handled by all stages as soon as they are produced, and guard of a stage can refer to attributes of distant stages. Thus this model is not directly executable on a distributed architecture. As highlighted in [18], distributed implementation may require restructuring the original GSM schema and relies on locking protocols to ensure that the outcome of the global execution is preserved.

This paper presents a declarative model for the specification of collaborative systems where the stakeholders interact according to an asynchronous message-based communication schema.

Case-management usually consists in assembling relevant information by calling *tasks*, which may in turn call subtasks. Case elicitation needs not be implemented as a sequence of successive calls to subtasks, and several subtasks can be performed in parallel. To allow as much concurrency as possible in the execution of tasks, we favor a *declarative* approach where task dependencies are specified without imposing a particular execution order.

Attribute grammars [28, 34] are particularly adapted to that purpose. The model proposed in this paper is a variant of attribute grammar, called *Guarded Attributed Grammar* (GAG). We use a notation reminiscent of unification grammars, and inspired by the work of Deransart and Maluszynski [15] relating attribute grammars with definite clause programs.

A production of a grammar is, as usual, described by a left-hand side, indicating a non-terminal to expand, and a right-hand side, describing how to expand this non-terminal. We furthermore interpret a production of the grammar as a way to decompose a task (the symbol in the left-hand side of the production) into sub-tasks associated with the symbols in its right-hand side. The semantics rules basically serve as a glue between the task and its sub-tasks by making the necessary connections between the corresponding inputs and outputs (associated respectively with inherited and synthesized attributes).

In this declarative model, the lifecycle of artifacts is left implicit. Artifacts under evaluation can be seen as incomplete structured documents, i.e., trees with *open nodes* corresponding to parts of the document that remain to be completed. Each open node is attached a so-called *form* interpreted as a task. A form consists of a task name together with some inherited attributes (data resulting from previous executions) and some synthesized attributes. The latter are variables subscribing to the values that will emerge from task execution.

Productions are *guarded* by patterns occurring at the inherited positions of the left-hand side symbol. Thus a production is enabled at an open node if the patterns match with the corresponding attribute values as given in the form. The evolution of the artifact thus depends both on previously computed data (stating which production is enabled) and the stakeholder's decisions (choosing a particular production amongst those which are enabled at a given moment, and inputting associated data). Thus GAGs are both *data-driven* and *user-centric*.

Data manipulated in guarded attributed grammars are of two kinds. First, the tasks communicate using *forms* which are temporary information used for communication purpose only, essentially for requesting values. Second, *artifacts* are structured documents that record the history of cases (log of the system). An artifact grows monotonically –we never erase information. Moreover, every part of the artifact is edited by a unique stakeholder –the owner of the corresponding nodes– hence avoiding edition conflicts. These properties are instrumental to obtain a simple and robust model that can easily be implemented on a distributed asynchronous architecture.

The modeling of a distributed collaborative system using GAG proceeds in two stages. First, we represent the workspaces of the various stakeholders as the collections of the artifacts they respectively handle. An artifact is a structured document with some active parts. Indeed, an open node is associated with a task that implicitly describes the data to be further substituted to the node. For that reason these workspaces are termed *active workspaces*. Second, we define collaborative systems by making the various user's active workspaces communicate with each other using asynchronous message passing.

This notion of *active documents* is close to the model of Active XML introduced by Abiteboul et al. [2] which consists of semi-structured documents with embedded service calls. Such an embedded service call is a query on another document, triggered when a corresponding guard is satisfied. The model of active documents can be distributed over a network of machines [1, 23]. This setting can be instancied in many ways, according to the formalism used for specifying the guards, the query language, and the class of documents. The model of guarded attribute grammars is close to this general schema with some differences: First of all, guards in GAGs apply to the attributes of a single node while guards in AXML are properties that can be checked on a complete document. The invocation of a service in AXML creates a temporary document (called the workspace) that is removed from the document when the service call returns. In GAGs, a rule applied to solve a task adds new children to the node, and all computations performed for a task are preserved in the artifact. This provides a kind of monotony to artifacts, an useful property for verification purpose.

The rest of the paper is organized as follows. Section 2 introduces the model of guarded attribute grammars and focuses on their use to standalone applications. The approach is extended in Section 3 to account for systems that call for external services. In this context we introduce a composition

of guarded attribute grammars. Some formal properties of guarded attribute grammars are studied in Section 4. Section 5 presents some notations and constructions allowing us to cope with the complexity of real-life systems. This specification language is illustrated in Section 6 on a case study for a disease surveillance system. Finally an assessment of the model and future research directions are given in conclusion.

2. GUARDED ATTRIBUTE GRAMMARS

This section is devoted to a presentation of the model of guarded attribute grammars. We start with an informal presentation that shows how the rules of a guarded attribute grammar can be used to structure the workspace of a stakeholder and formalize the workspace update operations. In two subsequent subsections we respectively define the syntax and the behavior of guarded attribute grammars. The section ends with basic examples used to illustrate some of the fundamental characteristics of the model.

2.1 A Grammatical Approach to Active Workspaces

Our model of collaborative systems is centered on the notion of user's workspace. We assume that the workspace of a user is given by a mindmap –simply call a *map* hereafter. It is a tree used to visualize and organize tasks in which the user is involved together with information used for the resolution of the tasks. The workspace of a given user may, in fact, consist of several maps where each map is associated with a particular *service* offered by the user. To simplify, one can assume that a user offers a unique service so that any workspace can be identified with its graphical representation as a map.

For instance the map shown in Fig. 1 might represent the workspace of a clinician acting in the context of a disease surveillance system.

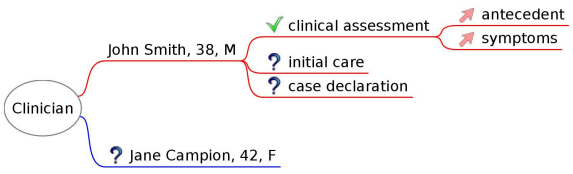


Figure 1: Active workspace of a clinician

The service provided by the clinician is to identify the symptoms of influenza in a patient, clinically examines the patient eventually placing him under therapeutic care, declaring the suspect cases to the Disease Surveillance Center (DCS), and monitoring the patient based on subsequent requests from the epidemiologist or the biologist.

Each call to this service, namely when a new patient comes to the clinician, creates a new tree rooted at the central node of the map. This tree is an *artifact* that represents a structured document for recording information about the patient all along being taken over in the system. Initially the artifact is reduced to a single (open) node that bears information about the name, age and sex of the patient. An open node, graphically identified by a question mark, represents a *pending task* that requires the attention of the clinician. In our example the initial task of an artifact is to clinically examine the patient. This task is refined into three subtasks: clinical assessment, initial care, and case declaration.

Our first goal is to ease the work of the clinician by avoiding a manual updating of the map. In order to automate transformations of the map we must first proceed to a classification of the different nodes –indicating their *sort*. Intuitively two open nodes are of the same sort when they can be refined by the same subtrees –they can have the same future. It then becomes possible, depending on the sort of an open node, to associate with it specific information –the *attributes* of the sort– and to specify in which way the node can be developed.

We interpret a task as a problem to be solved, that can be completed by refining it into sub-tasks using *business rules*. In a first approximation, a (business) rule can be modelled by a *production* $P : s_0 \to s_1 \cdots s_n$ expressing that task s_0 can be reduced to subtasks s_1 to s_n. For instance the production

$$\begin{array}{rl} \textbf{patient} \to & \textbf{clinical\_assessment} \\ & \textbf{initial\_care} \\ & \textbf{case\_declaration} \end{array}$$

states that a task of sort **patient**, the axiom of the grammar associated with the service provided by the clinician, can be refined by three subtasks whose sorts are respectively **clinical_assessment**, **initial_care**, and **case_declaration**.

If several productions with the same left-hand side s_0 exist then the choice of a particular production corresponds to a *decision* made by the user. For instance the clinician has to decide whether the case under investigation has to be declared to the Disease Surveillance Center or not. This decision can be reflected by the following two productions:

$$\begin{array}{lll} \text{suspect\_case} & : & \textbf{case\_declaration} \to \textbf{follow\_up} \\ \text{benign\_case} & : & \textbf{case\_declaration} \to \end{array}$$

If the case is reported as suspect then the clinician will have to follow up the case according to further requests of the biologist or of the epidemiologist. On the contrary, if the clinician has described the case as benign, it is closed with no follow up actions, More generally the tasks on the right-hand side of each production represent what remains to be done to resolve the task on the left-hand side in case this production is chosen.

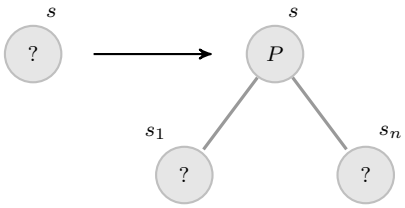
If P is the unique production having s_0 in its left-hand side, then there is no real decision to make and such a rule is interpreted as a logical decomposition of the task s_0 into substasks s_1 to s_n. Such a production will automatically be triggered without human intervention.

Accordingly, we model an artifact as a tree whose nodes are sorted. We write $X :: s$ to indicate that node X is of sort

s. An artifact is given by a set of equations of the form $X = P(X_1, \ldots, X_n)$, stating that $X :: s$ is a node labeled by production $P : s \to s_1 \cdots s_n$ and with successor nodes $X_1 :: s_1$ to $X_n :: s_n$. In that case node X is said to be a *closed* node defined by equation $X = P(X_1, \ldots, X_n)$ – we henceforth assume that we do not have two equations with the same left-hand side. A node $X :: s$ defined by no equation (i.e. that appears only in the right hand side of an equation) is an *open node*. It corresponds to a pending task of sort s.

The lifecycle of an artifact is implicitly given by the set of productions:

1. The artifact initially associated with a case is reduced to a single open node.
2. An open node X of sort s can be *refined* by choosing a production $P : s \to s_1 \ldots s_n$ that fits its sort.
   The open node X becomes a closed node –defined as $X = P(X_1, \ldots, X_n)$– under the decision of applying production P to it. In doing so task s associated with X is replaced by n subtasks s_1 to s_n and new open nodes $X_1 :: s_1$ to $X_n :: s_n$ are created accordingly.



3. The case has reached completion when its associated artifact is closed, i.e. it no longer contains open nodes.

Using the productions, the stakeholder can edit his workspace –the map– by selecting an open node –a pending task–, choosing one of the business rules that can apply to it, and inputting some values –information transmitted to the system.

However, plain context-free grammars are not sufficient to model the interactions and data exchanged between the various tasks associated with open nodes. For that purpose, we attach additional information to open nodes using *attributes*. Each sort $s \in S$ comes equipped with a set of *inherited* attributes and a set of *synthesized* attributes. Values of attributes are given by *terms* over a ranked alphabet. Recall that such a term is either a variable or an expression of the form $c(t_1, \ldots, t_n)$ where c is a symbol of rank n, and $t_1, \ldots, t_n$ are terms. In particular a constant c, i.e. a symbol of rank 0, will be identified with the term $c()$. We denote by $var(t)$ the set of variables used in term t.

Definition 2.1 (Forms). *A **form** of sort s is an expression $F = s(t_1, \ldots, t_n)\langle u_1, \ldots, u_m\rangle$ where $t_1, \ldots, t_n$ (respectively $u_1, \ldots, u_m$) are terms over a ranked alphabet — the alphabet of attribute's values— and a set of variables $var(F)$. Terms $t_1, \ldots, t_n$ give the values of the **inherited attributes** and $u_1, \ldots, u_m$ the values of the **synthesized attributes** attached to form F.*

(Business) rules are productions where sorts are replaced by forms of the corresponding sorts. More precisely, a rule is of the form

$$s_0(p_1, \ldots, p_n)\langle u_1, \ldots, u_m\rangle \to s_1(t_1^{(1)}, \ldots, t_{n_1}^{(1)})\langle y_1^{(1)}, \ldots, y_{m_1}^{(1)}\rangle$$
$$\vdots$$
$$s_k(t_1^{(k)}, \ldots, t_{n_k}^{(k)})\langle y_1^{(k)}, \ldots, y_{m_k}^{(k)}\rangle$$

where the p_i's, the u_j's, and the $t_j^{(\ell)}$'s are terms and the $y_j^{(\ell)}$'s are variables. The forms in the right-hand side of a rule are *tasks* given by forms

$$F = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle$$

where the synthesized positions are (distinct) variables $y_1, \ldots, y_m$ –i.e., they are not instantiated. The rationale is that we invoke a task by filling in the inherited positions of the form –the entries– and by indicating the variables that expect to receive the results returned during task execution –the *subscriptions*.

Any open node is thus attached to a task. The corresponding task execution is supposed *(i)* to construct the tree that will refine the open node and *(ii)* to compute the values of the synthesized attributes –i.e., it should return the subscribed values. A task is enacted by applying rules. More precisely, the rule can apply in an open node X when its left-hand side matches with task $s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ attached to node X. For that purpose the terms p_i's are used as patterns that should match the corresponding data d_i's. When the rules applies, new open nodes are created and they are respectively associated with the forms –tasks– in the right-hand side of the rule. The values of u_j's are then returned to the corresponding variables y_j's that subscribed to these values. For instance applying rule (see Fig. 2)

$$R : s_0(a(x_1, x_2))\langle b(y_1'), y_2'\rangle \to s_1(c(x_1))\langle y_1'\rangle \;\; s_2(x_2, y_1')\langle y_2'\rangle$$

to a node associated with tasks $s_0(a(t_1, t_2))\langle y_1, y_2\rangle$ gives rise to the substitution $x_1 = t_1$ and $x_2 = t_2$. The two newly-created open nodes are respectively associated with the tasks $s_1(c(t_1))\langle y_1'\rangle$ and $s_2(t_2, y_1')\langle y_2'\rangle$ and the values $b(y_1')$ and y_2' are substituted to the variables y_1 and y_2 respectively.

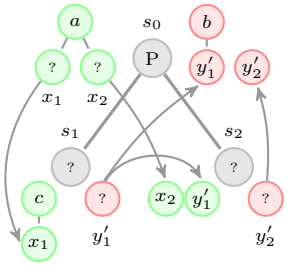


Figure 2: A business rule

This formalism puts emphasis on a declarative (logical) decomposition of tasks to avoid overconstrained schedules. Indeed, semantic rules and guards do not prescribe any ordering on task executions. Moreover ordering of tasks depend on the exchanged data and therefore are determined at run

time. In this way, the model allows as much concurrency as possible in the execution of the current pending tasks.

Furthermore the model can incrementally be designed by observing user's daily practice and discussing with her: We can initially let the user manually develops large parts of the map and progressively improve the automation of the process by refining the classification of the nodes –improving our knowledge on the ontology of the system– and introducing new business rules when recurrent patterns of activities are detected.

2.2 Guarded Attribute Grammars Syntax

Attribute grammars, introduced by Donald Knuth in the late sixties [28], have been instrumental in the development of syntax-directed transformations and compiler design. More recently this model has been revived for the specification of structured document's manipulations mainly in the context of web-based applications. The expression *grammareware* has been coined in [27] to qualify tools for the design and customization of grammars and grammar-dependent softwares. One such tool is the UUAG system developped by Swierstra and his group. They relied on purely functional implementations of attribute grammars [26, 36, 5] to build a domain specific languages (DSL) as a set of functional combinators derived from the semantic rules of an attribute grammar [16, 36, 35].

An attribute grammar is obtained from an underlying grammar by associating each sort s with a set $Att(s)$ of *attributes* —which henceforth should exist for each node of the given sort— and by associating each production $P : s \to s_1 \dots s_n$ with semantic rules describing the functional dependencies between the attributes of a node labelled P (hence of sort s) and the attributes of its successor nodes –of respective sorts s_1 to s_n. We use a non-standard notation for attribute grammars, inspired from [14, 15]. Let us introduce this notation on an example before proceeding to the formal definition.

Example 2.2 (Flattening of a binary tree).
Our first illustration is the classical example of the attribute grammar that computes the flattening of a binary tree, i.e., the sequence of the leaves read from left to right. The semantic rules are usually presented as shown in Fig. 2.2. The sort *bin* of binary trees has two attributes: The inherited attribute h contains an accumulating parameter and the synthesized attribute s eventually contains the list of leaves of the tree appended to the accumulating parameter. Which we may write as $t \cdot s = \mathit{flatten}(t) +\!\!+ t \cdot h$, i.e., $t \cdot s = \mathit{flat}(t, t \cdot h)$ where $\mathit{flat}(t, h) = \mathit{flatten}(t) +\!\!+ h$. The semantics rules stem from the identities:

$$\mathit{flatten}(t) = \mathit{flat}(t, Nil)$$
$$\mathit{flat}(\mathrm{Fork}(t_1, t_2), h) = \mathit{flat}(t_1, \mathit{flat}(t_2, h))$$
$$\mathit{flat}(\mathrm{Leaf}_a, h) = \mathrm{Cons}_a(h)$$

We present the semantics rules of Fig. 2.2 using the following syntax:

$$\begin{array}{lll} \mathrm{Root} & : & \mathit{root}()\langle x\rangle \to \mathit{bin}(Nil)\langle x\rangle \\ \mathrm{Fork} & : & \mathit{bin}(x)\langle y\rangle \to \mathit{bin}(z)\langle y\rangle\ \mathit{bin}(x)\langle z\rangle \\ \mathrm{Leaf}_a & : & \mathit{bin}(x)\langle \mathrm{Cons}_a(x)\rangle \to \end{array}$$

The *syntactic categories* of the grammar, also called its *sorts*, namely *root* and *bin* are associated with their inherited attributes –given as a list of arguments: $(t_1, \dots, t_n)$– and their

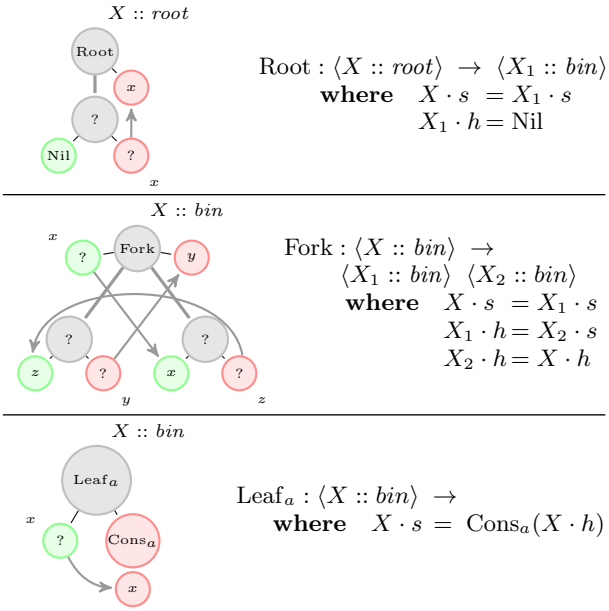


Figure 3: Flattening of a binary tree

synthesized attributes –the co-arguments:$\langle u_1, \dots, u_m\rangle$. A variable x is an *input variable*, denoted as $x^?$, if it appears in an inherited attribute in the left-hand side or in a synthesized attribute in the right-hand side. It corresponds to a piece of information stemming respectively from the context of the node or from the subtree rooted at the corresponding successor node. These variables should be pairwise distinct. Symmetrically a variable is an *output variable*, denoted as $x^!$, if it appears in a synthesized attribute of the left-hand side or in an inherited attribute of the right-hand side. It corresponds to values computed by the semantic rules and sent respectively to the context of the node or the subtree rooted at the corresponding successor node. Indeed, if we annotate the occurrences of variables with their polarity – input or output– one obtains:

$$\begin{array}{lll} \mathrm{Root} & : & \mathit{root}()\langle x^!\rangle \to \mathit{bin}(Nil)\langle x^?\rangle \\ \mathrm{Fork} & : & \mathit{bin}(x^?)\langle y^!\rangle \to \mathit{bin}(z^!)\langle y^?\rangle\ \mathit{bin}(x^!)\langle z^?\rangle \\ \mathrm{Leaf}_a & : & \mathit{bin}(x^?)\langle \mathrm{Cons}_a(x^!)\rangle \to \end{array}$$

And if we draw an arrow from the (unique) occurrence of $x^?$ to the (various) occurrences of $x^!$ for each variable x to witness the data dependencies then the above rules correspond precisely to the three figures shown on the left-hand side of Table 2.2. *End of Exple 2.2*

Guarded attribute grammars extend the traditional model of attribute grammars by allowing patterns rather than plain variables –as it was the case in the above example– to represent the inherited attributes in the left-hand side of a rule. Patterns allow the semantic rules to process by case analysis based on the shape of some of the inherited attributes, and in this way to handle the interplay between the data –contained in the inherited attributes– and the control –the enabling of rules.

Definition 2.3 (Guarded Attribute Grammars). *Given a set of sorts S with fixed inherited and synthesized attributes, a **guarded attribute grammar** (GAG) is a set of rules $R : F_0 \rightarrow F_1 \cdots F_k$ where the $F_i :: s_i$ are forms. A sort is* **used** *(respectively* **defined***) if it appears in the right-hand side (resp. the left-hand side) of some rule. A guarded attribute grammar G comes with a specific set of sorts* $\mathbf{axioms}(G) \subseteq \mathrm{def}(G) \setminus \mathrm{Use}(G)$ *–called the **axioms** of G– that are defined and not used. They are interpreted as the provided services. Sorts which are used but not defined are interpreted as external services used by the guarded attribute grammar. The values of the inherited attributes of left-hand side F_0 are called the **patterns** of the rule. The values of synthesized attributes in the right-hand side are variables. These occurrences of variables together with the variables occurring in the patterns are called the **input occurrences** of variables. We assume that each variable has **at most one input occurrence**.*

Remark 2.4. We have assumed in Def. 2.3 that axioms do not appear in the right-hand side of rules. This property will be instrumental to prove that strong-acyclicity –a property that guarantee a safe distribution of the GAG on an asynchronous architecture– can be compositionally verified. Nonetheless a specification that does not satisfy this property can easily be transformed into an equivalent specification that satisfies it: For each axiom s that occurs in the right-hand side of the rule we add a new symbol s' that becomes axioms in the place of s and we add copies of the rules associated with s –containing s in their left-hand side– in which we replace the occurrence of s in the left-hand side by s'. In this way we distinguish s used as a service by the environment of the GAG –role which is now played by s'– from its uses as an internal subtask –role played by s in the transformed GAG.

End of Remark 2.4

A rule of a GAG specifies the values at output positions –value of a synthesized attribute of s_0 or of an inherited attribute of $s_1, \ldots, s_n$. We refer to these correspondences as the **semantic rules**. More precisely, the inputs are associated with (distinct) variables and the value of each output is given by a term.

A variable can have several occurrences. First it may appear (once) as an input and it may also appear in output values. The corresponding occurrence is respectively said to be in an *input* or in an *output position*. One can define the following transformation on rules whose effect is to annotate each occurrence of a variable so that $x^?$ (respectively $x^!$) stands for an occurrence of x in an input position (resp. in an output position).

$$
\begin{aligned}
!(F_0 \rightarrow F_1 \cdots F_k) &= ?(F_0) \rightarrow !(F_1) \cdots !(F_k) \\
?(s(t_1, \ldots t_n)\langle u_1, \ldots u_m\rangle) &= s(?(t_1), \ldots ?(t_n))\langle !(u_1), \ldots !(u_m)\rangle \\
!(s(t_1, \ldots t_n)\langle u_1, \ldots u_m\rangle) &= s(!(t_1), \ldots !(t_n))\langle ?(u_1), \ldots ?(u_m)\rangle \\
?(c(t_1, \ldots t_n)) &= c(?(t_1), \ldots ?(t_n)) \\
!(c(t_1, \ldots t_n)) &= c(!(t_1), \ldots !(t_n)) \\
?(x) &= x^? \\
!(x) &= x^!
\end{aligned}
$$

The conditions stated in Definition 2.3 say that in the labelled version of a rule each variable occurs at most once in an input position, i.e., that $\{?(F_0), !(F_1), \ldots, !(F_k)\}$ is an admissible labelling of the set of forms in rule R according to the following definition.

Definition 2.5 (Link Graph). *A labelling in $\{?, !\}$ of the variables $var(\mathcal{F})$ of a set of forms $\mathcal{F}$ is **admissible** if the labelled version of a form $F \in \mathcal{F}$ is given by either $!F$ or $?F$ and each variable has at most one occurrence labelled with $?$. The occurrence $x^?$ identifies the place where the value of variable x is defined and the occurrences of $x^!$ identify the places where this value is used. The **link graph** associated with an admissible labelling of a set of forms $\mathcal{F}$ is the directed graph whose vertices are the occurrences of variables with an arc from v_1 to v_2 if these vertices are occurrences of a same variable x, labelled $?$ in v_1 and $!$ in v_2. This arc, depicted as follows,*

*means that the value produced in the **source vertex** v_1 should be forwarded to the **target vertex** v_2. Such an arc is called a **data link**.*

Definition 2.6 (Underlying Grammar). *The **underlying grammar** of a guarded attribute grammar G is the context-free grammar $\mathcal{U}(G) = (N, T, A, \mathcal{P})$ where*

- *the non-terminal symbols $s \in N$ are the defined sorts,*
- *$T = S \setminus N$ is the set of terminal symbols –the external services–,*
- *$A = \mathbf{axioms}(G)$ is the set of axioms of the guarded attribute grammar, and*
- *the set of productions $\mathcal{P}$ is made of the underlying productions $\mathcal{U}(R) : s_0 \rightarrow s_1 \cdots s_k$ of rules $R : F_0 \rightarrow F_1 \cdots F_k$ with $F_i :: s_i$.*

*A guarded attribute grammar is said to be **autonomous** when its underlying grammar contains no terminal symbols.*

Intuitively an autonomous guarded attribute grammar represents a standalone application: It corresponds to the description of particular services, associated with the axioms, whose realizations doe not rely on external services.

2.3 The Behavior of Autonomous Guarded Attribute Grammars

Attribute grammars are applied to input abstract syntax trees. These trees are usually produced by some parsing algorithm during a previous stage. The semantic rules are then used to decorate the node of the input tree by attribute values. In our setting, the generation of the tree and its evaluation using the semantic rules are intertwined since the input tree represents an artifact under construction. An artifact is thus an incomplete abstract syntax tree that contains closed and open nodes. A closed node is labelled by the rule that was used to create it. An open node is associated with a form that contains all the needed information for its further refinements. The information attached to an open node consists of the sort of the node and the current value of its attributes. The synthesized attributes of an open node are undefined and are thus associated with variables.

DEFINITION 2.7 (CONFIGURATION). *A **configuration** Γ of an autonomous guarded attribute grammar is an S-sorted set of nodes $X \in nodes(\Gamma)$ each of which is associated with a defining equation in one of the following form where $var(\Gamma)$ is a set of variables associated with Γ:*

Closed node: *$X = R(X_1, \ldots, X_k)$ where $X :: s$, and $X_i :: s_i$ for $1 \leq i \leq k$, and $\mathcal{U}(R) : s \to s_1 \ldots s_k$ is the underlying production of rule R. Rule R is the **label** of node X and nodes X_1 to X_n are its **successor nodes**.*

Open node: *$X = s(t_1, \ldots, t_n)\langle x_1, \ldots, x_m\rangle$ where X is of sort s and $t_1, \ldots, t_k$ are terms with variables in $var(\Gamma)$ that represent the values of the inherited attributes of X, and $x_1, \ldots, x_m$ are variables in $var(\Gamma)$ associated with its synthesized attributes.*

Each variable in $var(\Gamma)$ occurs at most once in a synthesized position. Otherwise stated $!\Gamma = \{!F \mid F \in \Gamma\}$ is an admissible labelling of the set of forms occurring in Γ. A node is called a **root node** *when its sort is an axiom. Each node is the successor of a unique node, called its **predecessor**, except for the root nodes that are the successor of no other nodes. Hence a configuration is a set of trees – abstract-syntax trees of the underlying grammar– which we call the **artifacts** of the configuration. Each axiom is associated with a* **map** *made of the artifacts of the corresponding sort. A map thus collects the artifacts corresponding to a specific service of the GAG.*

In order to specify the effect of applying a rule at a given node of a configuration (Definition 2.11) we first recall some notions about substitutions.

RECALL 2.8 (ON SUBSTITUTIONS). We identify a substitution σ on a set of variables $\{x_1, \ldots, x_k\}$, called the **domain** of σ, with a system of equations

$$\{x_i = \sigma(x_i) \mid 1 \leq i \leq k\}$$

The set $\mathrm{var}(\sigma) = \bigcup_{1 \leq i \leq k} \mathrm{var}(\sigma(x_i))$ of variables of σ, is disjoint from the domain $\mathrm{dom}(\sigma)$ of σ. Conversely a system of equations $\{x_i = t_i \mid 1 \leq i \leq k\}$ defines a substitution σ with $\sigma(x_i) = t_i$ if it is in **solved form**, i.e., none of the variables x_i appears in some of the terms t_j. In order to transform a system of equations $E = \{x_i = t_i \mid 1 \leq i \leq k\}$ into an equivalent system $\{x_i = t'_j \mid 1 \leq j \leq m\}$ in solved form one can iteratively replace an occurrence of a variable x_i in one of the right-hand side term t_j by its definition t_i until no variable x_i occurs in some t_j. This process terminates when the relation $x_i \succ x_j \Leftrightarrow x_j \in \mathrm{var}(\sigma(x_i))$ is acyclic. One can easily verify that, under this assumption, the resulting system of equation $SF(E) = \{x_i = t'_i \mid 1 \leq i \leq n\}$ in solved form does not depend on the order in which the variables x_i have been eliminated from the right-hand sides. When the above condition is met we say that the set of equations is **acyclic** and that it **defines** the substitution associated with the solved form. *End of Recall 2.8*

The composition of two substitutions σ, σ', where $\mathrm{var}(\sigma') \cap \mathrm{dom}(\sigma) = \emptyset$, is denoted by $\sigma\sigma'$ and defined by $\sigma\sigma' = \{x = t\sigma' | x = t \in \sigma\}$. Similarly, we let $\Gamma\sigma$ denote the configuration obtained from Γ by replacing the defining equation $X = F$ of each open node X by $X = F\sigma$.

We now define more precisely when a rule is enabled at a given open node of a configuration and the effect of applying the rule. First, note that variables of a rule are formal parameters whose scope is limited to the rule. They can injectively be renamed in order to avoid clashes with variables names appearing in the configuration. Therefore we always assume that the set of variables of a rule R is disjoint from the set of variables of configuration Γ when applying rule R at a node of Γ. As informally stated in the previous section, a rule R applies at an open node X when its left-hand side $s(p_1, \ldots, p_n)\langle u_1, \ldots u_m\rangle$ matches with the definition $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$, namely the task attached to X in Γ.

First, the patterns p_i should match with the data d_i according to the usual pattern matching given by the following inductive statements

$\mathbf{match}(c(p'_1, \ldots, p'_k), c'(d'_1, \ldots, d'_{k'}))$ with $c \neq c'$ fails
$\mathbf{match}(c(p'_1, \ldots, p'_k), c(d'_1, \ldots, d'_k)) = \sum_{i=1}^{k} \mathbf{match}(p'_i, d'_i)$
$\mathbf{match}(x, d) = \{x = d\}$

where the sum $\sigma = \sum_{i=1}^{k} \sigma_i$ of substitutions σ_i is defined and equal to $\bigcup_{i \in 1..k} \sigma_i$ when all substitutions σ_i are defined and associated with disjoint sets of variables. Note that since no variable occurs twice in the whole set of patterns p_i, the various substitutions $\mathbf{match}(p_i, d_i)$, when defined, are indeed concerned with disjoint sets of variables. Note also that $\mathbf{match}(c(), c()) = \emptyset$.

DEFINITION 2.9. *A form $F = s(p_1, \ldots, p_n)\langle u_1, \ldots u_m\rangle$ **matches** with a service call $F' = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ –of the same sort– when*

1. *the patterns p_i's matches with the data d_i's, defining a substitution $\sigma_{in} = \sum_{1 \leq i \leq n} \mathbf{match}(p_i, d_i)$,*
2. *the set of equations $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ is acyclic and defines a substitution σ_{out}.*

The resulting substitution $\sigma = \mathbf{match}(F, F')$ is given by $\sigma = \sigma_{out} \cup \sigma_{in}\sigma_{out}$.

REMARK 2.10. In most cases variables y_j do not appear in expressions d_i. And when it is the case one has only to check that patterns p_i's matches with data d_i's –substitution σ_{in} is defined– because then $\sigma_{out} = \{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ since the latter is already in solved form. Moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ because variables y_j do not appear in expressions $\sigma_{in}(x_i)$. *End of Remark 2.10*

DEFINITION 2.11 (APPLYING A RULE). *Let $R = F_0 \to F_1 \ldots F_k$ be a rule, Γ be a configuration, and X be an open node with definition $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ in Γ. We assume that R and Γ are defined over disjoint sets of variables. We say that R is* **enabled** *in X and write $\Gamma[R/X\rangle$, if the left-hand side of R matches with the definition of X. Then applying rule R at node X transforms configuration Γ into Γ', denoted as $\Gamma[R/X\rangle\Gamma'$, with Γ' defined as follows:*

$$\begin{array}{rcl} \Gamma' & = & \{X = R(X_1, \ldots, X_k)\} \\ & \cup & \{X_1 = F_1\sigma, \ldots, X_k = F_k\sigma\} \\ & \cup & \{X' = F\sigma \mid (X' = F) \in \Gamma \ \wedge \ X' \neq X\} \end{array}$$

where $\sigma = \mathbf{match}(F_0, X)$ and $X_1, \ldots, X_k$ are new nodes added to Γ'.

Thus the first effect of applying rule R to an open node X is that X becomes a closed node with label R and successor nodes X_1 to X_k. The latter are new nodes added to Γ'. They are associated respectively with the instances of the k forms in the right-hand side of R obtained by applying substitution σ to these forms. The definitions of the other nodes of Γ are updated using substitution σ –or equivalently σ_{out}. This update has no effect on the closed nodes because their defining equations in Γ contain no variable.

We conclude this section with two results justifying Definition 2.11. Namely, Prop. 2.12 states that if R is a rule enabled in a node X_0 of a configuration Γ with $\Gamma[R/X_0\rangle\Gamma'$ then Γ' is a configuration: Applying R cannot create a variable with several input occurrences. And Prop. 2.15 shows that substitution $\sigma = \mathbf{match}(F_0, X)$ resulting from the matching of the left-hand side $F_0 = s(p_1, \ldots, p_n)\langle u_1, \ldots, u_m\rangle$ of a rule R with the definition $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ of an open node X is the most general unifier of the set of equations $\{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\}$.

PROPOSITION 2.12. *If rule R is enabled in an open node X_0 of a configuration Γ and $\Gamma[R/X_0\rangle\Gamma'$ then Γ' is a configuration.*

PROOF. Let $R = F_0 \to F_1 \ldots F_k$ with left-hand side $F_0 = s(p_1, \ldots, p_n)\langle u_1, \ldots u_m\rangle$ and $X_0 = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ be the defining equation of X_0 in Γ. Since the values of synthesized attributes in the forms $F_1, \ldots, F_k$ are variables (by Definition 2.3) and since these variables are unaffected by substitution σ_{in} the synthesized attribute in the resulting forms $F_j\sigma_{in}$ are variables. The substitutions σ_{in} and σ_{out} substitute terms to the variables $x_1, \ldots, x_k$ appearing to the patterns and to the variables $y_1, \ldots, y_m$ respectively. Since x_i appears in an input position in R, it can appear only in an output position in the forms $!(F_1), \ldots !(F_k)$ and thus any variable of the term $\sigma_{in}(x_i)$ will appear in an output position in $!(F_i\sigma_{in})$. Similarly, since y_i appears in an input position in the form $!(s(u_1, \ldots, u_n)\langle y_1, \ldots, y_m\rangle)$, it can only appear in an output position in $!(F)$ for the others forms F of Γ. Consequently any variable of the term $\sigma_{out}(y_i)$ will appear in an output position in $!(F\sigma_{out})$ for any equation $X = F$ in Γ with $X \neq X_0$. It follows that the application of a rule cannot produce new occurrences of a variable in an input position and thus there cannot exist two occurrences $x^?$ of a same variable x in Γ'. *Q.E.D.*

Thus applying an enabled rule defines a binary relation on configurations.

DEFINITION 2.13. *A configuration Γ' is **directly accessible** from Γ, denoted by $\Gamma[\rangle\Gamma'$, whenever $\Gamma[R/X\rangle\Gamma'$ for some rule R enabled in node X of configuration Γ. Furthermore, a configuration Γ' is **accessible** from configuration Γ when $\Gamma[*\rangle\Gamma'$ where $[*\rangle$ is the reflexive and transitive closure of relation $[\rangle$.*

Recall that a substitution σ unifies a set of equations E if $t\sigma = t'\sigma$ for every equation $t = t'$ in E. A substitution σ is more general than a substitution σ', in notation $\sigma \leq \sigma'$, if $\sigma' = \sigma\sigma''$ for some substitution σ''. If a system of equations has a some unifier, then it has –up to an bijective renaming of the variables in σ– a *most general unifier*. In particular a set of equations of the form $\{x_i = t_i \mid 1 \leq i \leq n\}$ has a unifier if and only if it is acyclic. In this case, the corresponding solved form is its most general unifier.

RECALL 2.14 (ON UNIFICATION). We consider sets $E = E_? \uplus E_=$ containing equations of two kinds. An equation in $E_?$, denoted as $t \stackrel{?}{=} u$, represents a *unification goal* whose solution is a substitution σ such that $t\sigma = u\sigma$ –substitution σ unifies terms t and u. $E_=$ contains equations of the form $x = t$ where variable x occurs only there, i.e., we do not have two equations with the same variable in their left-hand side and such a variable cannot either occur in any right-hand side of an equation in $E_=$. A *solution* to E is any substitution σ whose domain is the set of variables occurring in the right-hand sides of equations in $E_=$ such that the compound substitution made of σ and the set of equations $\{x = t\sigma \mid x = t \in E_=\}$ unifies terms t and u for any equation $t \stackrel{?}{=} u$ in $E_?$. Two systems of equations are said to be *equivalent* when they have the same solutions. A *unification problem* is a set of such equations with $E_= = \emptyset$, i.e., it is a set of unification goals. On the contrary E is said to be in *solved form* if $E_? = \emptyset$, thus E defines a substitution which, by definition, is the most general solution to E. Solving a unification problem E consists in finding an equivalent system of equations E' in solved form. In that case E' is a *most general unifier* for E.
Martelli and Montanari Unification algorithm [30] proceeds as follows. We pick up non deterministically one equation in $E_?$ and depending on its shape apply the corresponding transformation:

1. $c(t_1, \ldots, t_n) \stackrel{?}{=} c(u_1, \ldots, u_n)$: replace it by equations $t_1 \stackrel{?}{=} u_1, \ldots, t_1 \stackrel{?}{=} u_1$.
2. $c(t_1, \ldots, t_n) \stackrel{?}{=} c'(u_1, \ldots, u_m)$ with $c \neq c'$: halt with failure.
3. $x \stackrel{?}{=} x$: delete this equation.
4. $t \stackrel{?}{=} x$ where t is not a variable: replace this equation by $x \stackrel{?}{=} t$.
5. $x \stackrel{?}{=} t$ where $x \notin \mathrm{var}(t)$: replace this equation by $x = t$ and substitute x by t in all other equations of E.
6. $x \stackrel{?}{=} t$ where $x \in \mathrm{var}(t)$ and $x \neq t$: halt with failure.

The condition in (5) is the occur check. Thus the computation fails either if the two terms of an equation cannot be unified because their main constructors are different or because a potential solution of an equation is necessarily an infinite tree due to a recursive statement detected by the occur check. System E' obtained from E by applying one of these rules, denoted as $E \Rightarrow E'$, is clearly equivalent to E. We iterate this transformation as long as we do not encounter a failure and some equation remains in $E_?$. It can be proved that all these computations terminate and either the original unification problem E has a solution –a unifier– and every computation terminates –and henceforth produces a solved set equivalent to E describing a most general unifier

of $E-$ or E has no unifier and every computation fails. We let

$$\sigma = \mathbf{mgu}(\{t_i = u_i\}_{1\le i\le n}) \ \textbf{iff} \ \left\{t_i \stackrel{?}{=} u_i\right\}_{1\le i\le n} \Rightarrow^* \sigma$$

End of Recall 2.14

Note that (5) and (6) are the only rules that can be applied to solve a unification problem of the form $\{y_i \stackrel{?}{=} u_i \mid 1 \le i \le n\}$, where the y_i are distinct variables. The most general unifier exists when the occur check always holds, i.e., rule (5) always applies. The computation amounts to iteratively replacing an occurrence of a variable y_i in one of the right-hand side term u_j by its definition u_i until no variable y_i occurs in some u_j, i.e., (see Recall 2.8) when this system of equation is acyclic. Hence any acyclic set of equations $\{y_i = u_i \mid 1 \le i \le n\}$ defines the substitution $\sigma = \mathbf{mgu}(\{y_i = u_i \mid 1 \le i \le n\})$.

PROPOSITION 2.15. *If $F_0 = s_0(p_1, \ldots, p_n)\langle u_1, \ldots, u_m\rangle$, left-hand side of a rule R, matches with the definition $X = s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ of an open node X then substitution $\sigma = \mathbf{match}(F_0, X)$ is the most general unifier of the set of equations $\{p_i = d_i \mid 1 \le i \le n\} \cup \{y_j = u_j \mid 1 \le j \le m\}$.*

PROOF. If a rule R of left-hand side $s_0(p_1, \ldots, p_n)\langle u_1, \ldots u_m\rangle$ is triggered in node $X_0 = s_0(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ then by Def. 2.11 $\left\{p_i \stackrel{?}{=} d_i\right\}_{1\le i\le n} \cup \left\{y_j \stackrel{?}{=} u_j\right\}_{1\le j\le m} \Rightarrow^* \sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j\sigma_{in}\right\}_{1\le j\le m}$ using only the rules (1) and (5). Now, by applying iteratively rule (5) one obtains

$$\sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j\sigma_{in}\right\}_{1\le j\le m} \Rightarrow^* \sigma_{in} \cup \mathbf{mgu}\,\{y_j = u_j\sigma_{in}\}_{1\le j\le m}$$

when the set of equations $\{y_j = u_j\sigma_{in}\}_{1\le j\le m}$ satisfies the occur check. Then $\sigma_{in} + \sigma_{out} \Rightarrow^* \sigma$ again by using rule (5). *Q.E.D.*

REMARK 2.16. The converse of Prop. 2.15 does not hold. Namely, one shall not deduce from Proposition 2.15 that the relation $\Gamma[R/X_0\rangle\Gamma'$ is defined whenever the left-hand side $\mathrm{lhs}(R)$ of R can be unified with the definition $\mathrm{def}(X_0, \Gamma)$ of X_0 in Γ with

$$\begin{array}{rcl} \Gamma' & = & \{X_0 = R(X_1, \ldots, X_k)\} \\ & \cup & \{X_1 = F_1\sigma, \ldots, X_k = F_k\sigma\} \\ & \cup & \{X = F\sigma \mid (X = F) \in \Gamma \ \wedge \ X \neq X_0\} \end{array}$$

where $\sigma = \mathbf{mgu}(\mathrm{lhs}(R), \mathrm{def}(X_0, \Gamma))$, and $X_1, \ldots, X_k$ are new nodes added to Γ'. Indeed, when unifying $\mathrm{lhs}(R)$ with $\mathrm{def}(X_0, \Gamma)$ one may generate an equation of the form $x = t$ where x is a variable in an inherited data d_i and t is an instance of a corresponding subterm in the associated pattern p_i. This would correspond to a situation where information is sent to the context of a node through one of its inherited attribute! Stated differently, with this alternative definition some parts of the pattern p_i could actually be used to filter out the incoming data value d_i while some other parts of the same pattern would be used to transfert synthesized information to the context. *End of Remark 2.16*

2.4 Some Examples

In this section we illustrate the behaviour of guarded attribute grammars with two examples. Example 2.17 describes an execution of the attribute grammar of Example 2.2. The specification in Example 2.2 is actually an ordinary attribute grammar because the inherited attributes in the left-hand sides of rules are plain variables. This example shows how data are lazily produced and send in push mode through attributes. It also illustrates the role of the data links and their dynamic evolutions. Example 2.18 illustrates the role of the guards by describing two processes acting as coroutines. The first process sends forth a list of values to the second process and it waits for an acknowledgement for each message before sending the next one.

EXAMPLE 2.17 (EXAMPLE 2.2 CONTINUED). Let us consider the attribute grammar of Example 2.2 and the initial configuration $\Gamma_0 = \{X_0 = root()\langle x\rangle, Y_0 = toor(x)\langle\rangle\}$ shown next

The annotated version $!\Gamma_0 = \{!F \mid F \in \Gamma_0\}$ of configuration Γ_0 is

$$!\Gamma_0 = \left\{X_0 = root()\langle x^?\rangle, Y_0 = toor(x^!)\langle\rangle\right\}$$

The data link from $x^?$ to $x^!$ says that the list of the leaves of the tree –that will stem from node X_0– to be synthesized at node X_0 should be forwarded to the inherited attribute of Y_0.

This tree is not defined in the intial configuration Γ_0. One can start developping it by applying rule Root : $root()\langle u\rangle \to bin(Nil)\langle u\rangle$ at node X_0 :: *root*. Actually the left-hand side $root()\langle u\rangle$ of rule Root matches with the definition $root()\langle x\rangle$ of X_0 with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = u\}$. Thus $\Gamma_0[\mathrm{Root}/X_0\rangle\Gamma_1$ where the annotated configuration $!\Gamma_1$ is given in Figure 4.

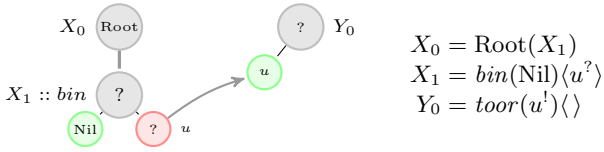


Figure 4: Configuration Γ_1

Note that substitution $\sigma_{\mathrm{out}} = \{x = u\}$ replaces the data link $(x^?, x^!)$ by a new link $(u^?, u^!)$ with the same target and whose source has been moved from the synthesized attribute of X_0 to the synthesized attribute of X_1.

The tree may be refined by applying rule

$$\mathrm{Fork} : bin(x)\langle y\rangle \to bin(z)\langle y\rangle \ bin(x)\langle z\rangle$$

at node $X_1 :: bin$ since its left-hand side $bin(x)\langle y\rangle$ matches with the definition $bin(\text{Nil})\langle u\rangle$ of X_1 with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{u = y\}$. Hence $\Gamma_1[\text{Fork}/X_1\rangle\Gamma_2$ where $!\Gamma_2$ is given in Figure 5.

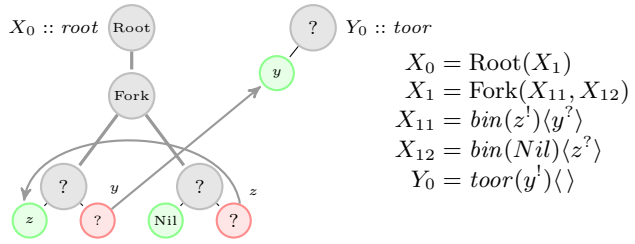


Figure 5: Configuration Γ_2

Rule $\text{Leaf}_c : bin(x)\langle\text{Cons}_c(x)\rangle \to$ applies at node X_{12} since its left-hand side $bin(x)\langle\text{Cons}_c(x)\rangle$ matches with the definition $bin(Nil)\langle z\rangle$ of X_{12} with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{z = \text{Cons}_c(\text{Nil})\}$. Hence $\Gamma_2[\text{Leaf}_c/X_{12}\rangle\Gamma_3$ where the annotated configuration $!\Gamma_3$ is given in Figure 6.

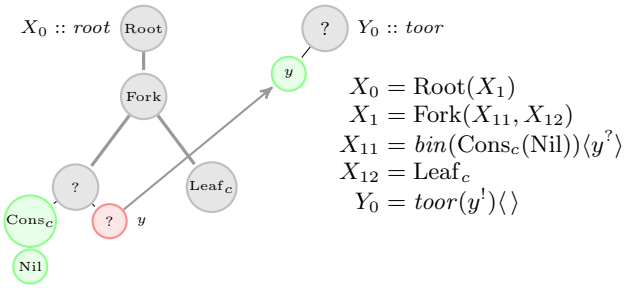


Figure 6: Configuration Γ_3

As a result of substitution $\sigma_{out} = \{z = \text{Cons}_c(\text{Nil})\}$ the value $\text{Cons}_c(\text{Nil})$ is transmitted through the link $(z^?, z^!)$ and this link disappears.

Rule Fork : $bin(x)\langle u\rangle \to bin(z)\langle u\rangle\ bin(x)\langle z\rangle$ may apply at node X_{11}: its left-hand side $bin(x)\langle u\rangle$ matches with the definition $bin(\text{Cons}_c(\text{Nil}))\langle y\rangle$ of X_{11} with $\sigma_{in} = \{x = \text{Cons}_c(\text{Nil})\}$ and $\sigma_{out} = \{y = u\}$. Hence $\Gamma_3[\text{Fork}/X_1\rangle\Gamma_4$ with configuration $?\Gamma_4$ given in Figure 7.

Rule $\text{Leaf}_a : bin(x)\langle\text{Cons}_a(x)\rangle \to$ applies at node X_{111} since its left-hand side $bin(x)\langle\text{Cons}_a(x)\rangle$ matches with the definition $bin(z)\langle u\rangle$ of X_{111} with $\sigma_{in} = \{x = z\}$ and $\sigma_{out} = \{u = \text{Cons}_a(z)\}$. Hence $\Gamma_4[\text{Leaf}_a/X_{111}\rangle\Gamma_5$ with configuration $!\Gamma_5$ given in Figure 8.

Using substitution $\sigma_{out} = \{u = \text{Cons}_a(z)\}$ the data $\text{Cons}_a(z)$ is transmitted through the link $(u^?, u^!)$ which, as a result, disappears. A new link $(z^?, z^!)$ is created so that the rest of the list, to be synthesized in node X_{112} can later be forwarded to the inherited attribute of Y_0.

Finally one can apply rule $\text{Leaf}_b : bin(x)\langle\text{Cons}_a(x)\rangle \to$ at node X_{112} since its left-hand side matches with the definition

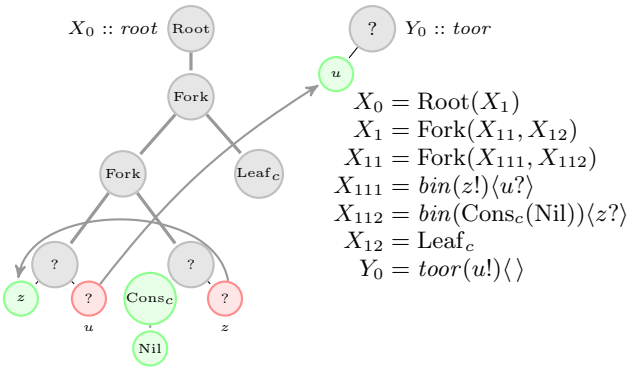


Figure 7: Configuration Γ_4

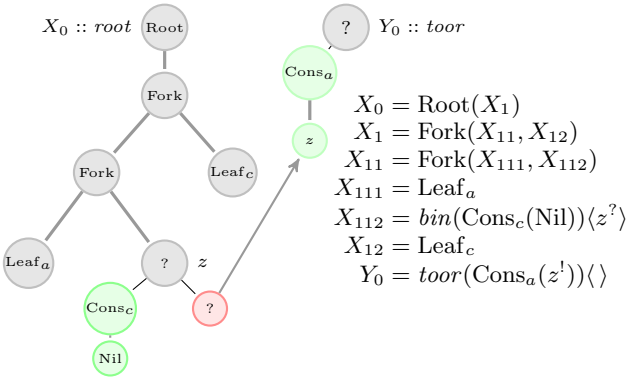


Figure 8: Configuration Γ_5

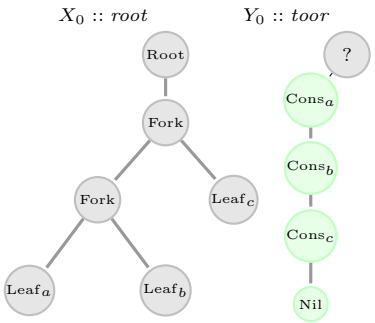
$bin(\text{Cons}_c(\text{Nil}))\langle z\rangle$ of X_{112} with $\sigma_{in} = \{x = \text{Cons}_c(\text{Nil})\}$ and $\sigma_{out} = \{z = \text{Cons}_b(\text{Cons}_c(\text{Nil}))\}$.

Therefore $\Gamma_5[\text{Leaf}_b/X_{112}\rangle\Gamma_6$ with configuration $!\Gamma_6$ given in Figure 9.

Now the tree rooted at node X_0 is closed –and thus it no longer holds attributes– and the list of its leaves has been entirely forwarded to the inherited attribute of node Y_0. Note that the recipient node Y_0 could have been refined in parallel with the changes of configurations just described.

End of Exple 2.17

The above example shows that data links are used to transmit data in push mode from a source vertex v –the input occurrence $x^?$ of a variable x– to some target vertex v' – an output occurrence $x^!$ of the same variable. These links $(x^!, x^?)$ are *transient* in the sense that they disappear as soon as variable x gets defined by the substitution σ_{out} induced by the application of a rule in some open node of the current configuration. If $\sigma_{out}(x)$ is a term t, not reduced to a variable, with variables $x_1, \ldots, x_k$ then vertex v' is refined by the term $t[x_i^!/x_i]$ and new vertices v'_i —associated with these new occurrences of x_i in an output position— are created. The original data link $(x^?, x^!)$ is replaced by all the corresponding instances of $(x_i^?, x_i^!)$. Consequently, a target is replaced by new targets which are the recipients for the subsequent pieces of information –maybe none because no new links are created when t contains no variable. If the term t



$$
\begin{aligned}
X_0 &= \mathrm{Root}(X_1)\\
X_1 &= \mathrm{Fork}(X_{11}, X_{12})\\
X_{11} &= \mathrm{Fork}(X_{111}, X_{112})\\
X_{111} &= \mathrm{Leaf}_a\\
X_{112} &= \mathrm{Leaf}_b\\
X_{12} &= \mathrm{Leaf}_c\\
Y_0 &= \mathit{toor}(\mathrm{Cons}_a(\mathrm{Cons}_b(\mathrm{Cons}_c(\mathrm{Nil}))))\langle\,\rangle
\end{aligned}
$$

Figure 9: Configuration Γ_6

is a variable y then the link $(x^?, x^!)$ is replaced by the link $(y^?, y^!)$ with the same target and whose source, the (unique) occurrence $x^?$ of variable x, is replaced by the (unique) occurrence $y^?$ of variable y. Therefore the direction of the flow of information is in both cases preserved: Channels can be viewed as "generalized streams" –that can fork or vanish– through which information is pushed incrementally.

EXAMPLE 2.18. Figure 10 shows a guarded attribute grammar that represents two coroutines communicating through lazy streams. Each process alternatively sends and receives data. More precisely the second process sends an acknowlegment –message $?b$– upon reception of a message sent by the left process. Initially or after reception of an acknowlegment of its previous message the left process can either send a new message or terminate the communication.

Production $!a : q_1(x')\langle a(y')\rangle \leftarrow q_2(x')\langle y'\rangle$ applies at node X_1 of configuration

$$\Gamma_1 = \left\{X = X_1 \| X_2,\ \ X_1 = q_1(x)\langle y\rangle,\ \ X_2 = q_2'(y)\langle x\rangle\right\}$$

shown in Figure 11 because its left-hand side $q_1(x')\langle a(y')\rangle$ matches with the definition $q_1(x)\langle y\rangle$ of X_1 with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = a(y')\}$. We get configuration

$$\Gamma_2 \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ \ X_1 = !a(X_{11}),\\ X_2 = q_2'(a(y'))\langle x\rangle,\ \ X_{11} = q_2(x)\langle y'\rangle \end{array} \right\}$$

shown on the middle of Figure 11.

Production $?a : q_2'(a(y))\langle x'\rangle \leftarrow q_1'(y)\langle x'\rangle$ applies at node X_2 of Γ_2 because its left-hand side $q_2'(a(y))\langle x'\rangle$ matches with the definition $q_2'(a(y'))\langle x\rangle$ of X_2 with $\sigma_{in} = \{y = y'\}$ and $\sigma_{out} = \{x = x'\}$. We get configuration

$$\Gamma_3 \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ \ X_1 = !a(X_{11}),\ \ X_2 = ?a(X_{21}),\\ X_{11} = q_2(x')\langle y'\rangle,\ X_{21} = q_1'(y')\langle x'\rangle\ \} \end{array} \right\}$$

shown on the right of Figure 11.

The corresponding acknowlegment may be sent and received leading to configuration

$$\Gamma_5 = \Gamma \cup \left\{X_{111} = q_1(x)\langle y\rangle,\ \ X_{211} = q_2'(y)\langle x\rangle\right\}.$$

$$\textit{where}\ \ \Gamma \ = \ \left\{ \begin{array}{l} X = X_1 \| X_2,\ X_1 = !a(X_{11}),\ X_2 = ?a(X_{21}),\\ X_{21} = !b(X_{211}),\ \ X_{11} = ?b(X_{111}) \end{array} \right\}.$$

The process on the left may decide to end communication by applying production $!\mathrm{stop} : q_1(x')\langle \mathrm{stop}\rangle \leftarrow$ at X_{111} with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = \mathrm{stop}\}$ leading to configuration

$$\Gamma_6 = \Gamma \cup \left\{X_{111} = !\mathrm{stop},\ \ X_{211} = q_2'(\mathrm{stop})\langle x\rangle\right\}.$$

The reception of this message by the process on the right corresponds to applying production $?\mathrm{stop} : q_2'(\mathrm{stop})\langle y\rangle \leftarrow$ at X_{211} with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = y\}$ leading to configuration

$$\Gamma_7 = \Gamma \cup \{X_{111} = !\mathrm{stop},\ \ X_{211} = ?\mathrm{stop}\}.$$

Note that variable x appears in an input position in Γ_6 and has no corresponding output occurrence. This means that the value of x is not used in the configuration. When production $?\mathrm{stop}$ is applied in node X_{211} variable y is substituted to x. Variable y has an output occurrence in production $?\mathrm{stop}$ and no input occurence meaning that the corresponding output attribute is not defined by the semantic rules. As a consequence this variable simply disappears in the resulting configuration Γ_7. If variable x was used in Γ_6 then the output occurrences of x would have been replaced by (output occurrences) of variable y that will remain undefined –no value will be substituted to y in subsequent transformations– until these occurrences of variables may possibly disappear.

End of Exple 2.18

3. COMPOSITION OF GAG

In this section we define the behavior of potentially non-autonomous guarded attributed grammars to account for systems that call for external services: A guarded attribute grammar providing some set of services may contain terminal symbols, namely symbols that do not occur in the left-hand sides of rules. These terminal symbols are interpreted as calls to external services that are associated with some other guarded attribute grammar. We introduce a composition of guarded attribute grammars and show that the behavior of the composite guarded attribute grammar can be recovered from the behavior of its components.

Recall that the behavior of an active workspace is given by a guarded attribute grammar G. Its configuration is a set of maps associated with each of the axioms, or services, of the grammar. A map contains the artifacts generated by calls to the corresponding service. We assume that each active workspace has a namespace $ns(G)$ used for the nodes X of its configuration, the variables x occurring in the values of attributes of these nodes, and also for references to variables belonging to others active workspaces –its subscriptions. Hence we have a name generator that produces unique identifiers for each newly created variable of the configuration. Furthermore, we assume that the name of a variable

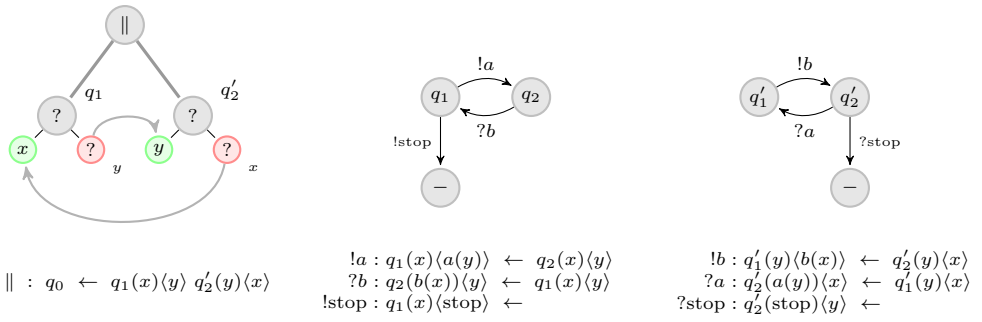


Figure 10: Coroutines with lazy streams

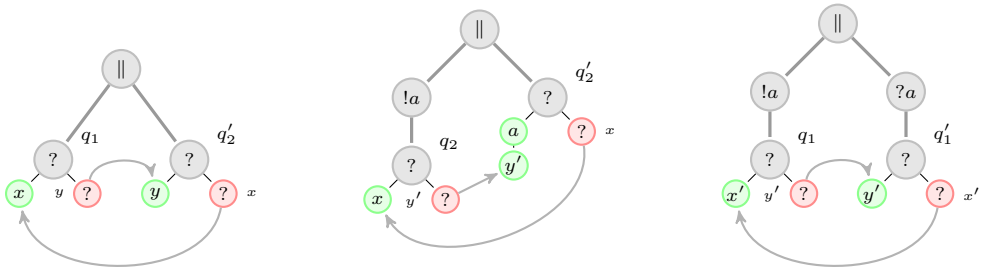


Figure 11: $\Gamma_1[!a/X_1\rangle\Gamma_2[?a/X_2\rangle\Gamma_3$

determines its *location*, namely the active workspace it belongs to. A configuration is given by a set of equations as stated in Definition 2.7 with the following changes.

1. A node associated with a terminal symbol is associated with no equation –it corresponds a service call that initiates an artifact in another active workspace.
2. We have equations of the form $y = x$ stating that distant variable y subscribes to the value of local variable x.
3. We have equations of the form $Y = X$ where Y is the distant node that created the artifact rooted at local node X. Hence X is a root node.

Futhermore we add an input and an output buffers to each configuration Γ. They contains messages respectively received from and send to distant locations. A message in the input buffer $in(\Gamma)$ is one of the following types.

1. $Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle$ tells that distant node Y calls service $s \in \textbf{axioms}(G)$. When reading this message we create a new root node X —the root of the artifact associated with the service call. And values $t_1, \ldots, t_n$ are assigned to the inherited attributes of node X while the distants variables $y_1, \ldots, y_m$ subscribe to the values of its synthesized attributes. We replace variable Y by a dummy variable (wildcard: _) when this service call is not invoked from a distant active workspace but from an external user of the system.
2. $x = t$ tells that local variable x receives the value t from a subscription created at a distant location.
3. $y = x$ states that distant variable y subscribes to the value of local variable x.

Symmetrically, a message in the output buffer $out(\Gamma)$ is one of the following types.

1. $X = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle$ tells that local node X calls the external service s –a terminal symbol– with values $t_1, \ldots, t_n$ assigned to the inherited attributes. And the local variables $y_1, \ldots, y_m$ subscribe to the values of the synthesized attributes of the distant node where the artifact generated by this service call will be rooted at.
2. $y = t$ tells that value t is sent to distant variable y according to a subscription made for this variable.
3. $x = y$ states that local variable x subscribes to the value of distant variable y.

The behavior of a guarded attribute grammar is given by relation $\Gamma \xrightarrow[M]{e} \Gamma'$ stating that event e transforms configuration Γ into Γ' and adds the set of messages M to the output buffer. An event is the application of a rule R to a node X of configuration Γ or the consumption of a message from its input buffer. Let us start with the former kind of event: $e = R/X$. let $X = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$ and $R = F \to F_1 \cdots F_k$ be a rule whose left-hand side matches with X producing substitution $\sigma = \textbf{match}(F, X)$. All variables occurring in the definition of node X are local. We recall that in order to avoid name clashes we rename all the variables of rule R with fresh names. We use the local name generator for that purpose. Hence all variables of R are also local variables –freshly created ones. Therefore all variables

used and defined by substitution σ are local variables. Then we let $\Gamma \xrightarrow[M]{R/X} \Gamma'$ where

$$\begin{array}{rcl}
\Gamma' & = & \{X = R(X_1, \ldots, X_k)\} \\
& \cup & \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in N\} \\
& \cup & \{X' = F\sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \\
& \cup & \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ is a variable}\} \\
& \cup & \{Y = X \mid (Y = X) \in \Gamma\} \\
M & = & \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in T\} \\
& \cup & \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ not a variable}\}
\end{array}$$

where $X_1, \ldots, X_k$ are new names in $ns(G)$. Note that when a distant variable y subscribes to some synthesized attribute of node X, namely $(y = y_i) \in \Gamma$, two situations can occur depending on whether $y_j\sigma$ is a variable or not. When $y_j\sigma = x$ is a (local) variable the subcription $y = y_i$ is replaced by subcription $y = x$: Variable y_i delegates the production of the required value to x. This operation is totally transparent to the location that initiated the subscription. But as soon as some value is produced – $y_j\sigma$ is not a variable – it is immediately sent to the subscribing variable even when this value contains variables: Values are produced and send incrementally.

Let us now consider the event associated with the consumption of a message $m \in out(\Gamma)$ in the output buffer.

1. If $m = (Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_q\rangle)$ then
$$\begin{array}{rl}
\Gamma' = \Gamma \cup & \{\overline{Y} = s(\overline{t}_1, \ldots, \overline{t}_n)\langle \overline{y}_1, \ldots, \overline{y}_q\rangle\} \\
\cup & \{y_j = \overline{y_j} \mid 1 \leq j \leq q\} \cup \{Y = \overline{Y}\}
\end{array}$$
where $\overline{Y}$, the variables $\overline{x}$ for $x \in var(t_i)$ and the variables $\overline{y}_j$ are new names in $ns(G)$, $\overline{t} = t[\overline{x}/x]$, and $M = \{\overline{x} = x \mid x \in var(t_i) \ \ 1 \leq i \leq n\}$.
2. If $m = (x = t)$ then $\Gamma' = \Gamma[x = t[\overline{y}/y]]$ where $\overline{y}$ are new names in $ns(G)$ associated with the variables y in t and $M = \{\overline{y} = y \mid y \in var(t)\}$.
3. If $m = (y = x)$ then $\Gamma' = \Gamma \cup \{y = x\}$ and $M = \emptyset$.

We now define the guarded attribute grammar resulting from the composition of a set of smaller guarded attribute grammars.

Definition 3.1 (Composition of GAG).
Let $G_1, \ldots, G_p$ be guarded attribute grammars with disjoint sets of non terminal symbols such that each terminal symbol of a grammar G_i that belongs to another grammar G_j must be an axiom of the latter: $T_i \cap S_j = T_i \cap \mathbf{axioms}(G_j)$ where $s \in T_i \cap \mathbf{axioms}(G_j)$ means that grammar G_i uses service s of G_j. Their **composition**, *denoted as $G = G_1 \oplus \cdots \oplus G_p$, is the guarded attribute grammar whose set of rules is the union of the rules of the G_is and with set of axioms $\mathbf{axioms}(G) = \cup_{1 \leq i \leq p}\mathbf{axioms}(G_j)$. We say that the G_i are the* **local grammars** *and G the* **global grammar** *of the composition. If some axiom of the resulting global grammar calls itself recursively we apply the transformation described in Rem. 2.4.*

One may also combine this composition with a restriction operator, $G \downarrow_{\mathbf{ax}}$, if the global grammar offers only a subset $\mathbf{ax} \subseteq \cup_{1 \leq i \leq p}\mathbf{axioms}(G_j)$ of the services provided by the local grammars.

Note that the set of terminal symbols of the composition is given by

$$T = (\cup_{1 \leq i \leq p} T_i) \setminus (\cup_{1 \leq i \leq p}\mathbf{axioms}(G_j))$$

i.e., its set of external services are all external services of a local grammar but those which are provided by some other local grammar. Note also that the set of non-terminal symbols of the global grammar is the (disjoint) union of the set of non-terminal symbols of the local grammars: $N = \cup_{1 \leq i \leq p} N_i$. This partition can be used to retrieve the local grammar by taking the rules of the global grammar whose sorts in their left-hand side belongs to the given equivalent class. Of course not every partition of the set of non-terminal symbols of a guarded attribute grammar corresponds to a decomposition into local grammars. To decompose a guarded attribute grammar into several components one can proceed as follows:

1. Select a partition $\mathbf{axiom}(G) \subseteq \cup_{1 \leq i \leq n}\mathbf{axioms}_i$ of the set of axioms. These sets are intended to represent the services associated with each of the local grammars.
2. Construct the local grammar G_i associated with services $\mathbf{axioms}_i$ by first taking the rules whose left-hand sides are forms of sort $s \in \mathbf{axioms}_i$, and then iteratively adding to G_i the rules whose left-hand sides are forms of sort $s \in N \setminus \cup_{j \neq i}\mathbf{axioms}_i$ such that s appears in the right-hand side of a rule previously added to G_i.
3. If appropriate, namely when a same rule is copied in several components, rename the non-terminal symbols of the local grammars to ensure that they have disjoint sets of non-terminal symbols.

The above transformation can duplicate rules in G in the resulting composition $\overline{G} = G_1 \oplus \cdots \oplus G_n$ but does not radically change the original specification.

Configurations of guarded attribute grammars are enriched with subscriptions –equations of the form $y = x$ – to enable communication between the various active workspaces. One might dispense with equations of the form $Y = X$ in the operational semantics of guarded attribute grammars. But they facilitate the proof of correctness of this composition (Prop. 3.2) by easing the reconstruction of the global configuration from its set of local configurations. Indeed, the global configuration can be recovered as $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ where operator $\oplus$ consists in taking the union of the systems of equations given as arguments and simplifying the resulting system by elimination of the copy rules: We drop each equation of the form $Y = X$ (respectively $y = x$) and replace each occurrence of Y by X (resp. of y by x).

Let $G = G_1 \oplus \cdots \oplus G_p$ be a composition , Γ_i be a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ the corresponding global configuration. Since the location of a variable derives from its identifier we know the location of destination of every message in the output buffer of a component. If M is a set of messages we let $M_i \subseteq M$ denote the set of messages in M to be forwarded to G_i. Their union

$M_\iota = \cup_{1\leq i\leq p} M_i$ is the set of *internal* messages that circulate between the local grammars. The rest $M_G = M \setminus M_\iota$ is the set of messages that remain in the output buffer of the global grammar –the *global* messages.

The join dynamics of the local grammars can be derived as follows from their individual behaviors, where e stands for R/X or a message m:

1. If $\Gamma_i \xrightarrow[M]{e} \Gamma_i'$ then $\Gamma \xRightarrow[M]{e} \Gamma'$ with $\Gamma_j = \Gamma_j'$ for $j \neq i$.

2. If $\Gamma \xRightarrow[M]{e} \Gamma'$ and $\Gamma' \xRightarrow[M']{m} \Gamma''$ for $m \in M$ then

$$\Gamma \xRightarrow[M\setminus\{m\}\cup M']{e} \Gamma''.$$

The correctness of the composition is given by the following proposition. It states that *(i)* every application of a rule can immediately be followed by the consumption of the local messages generated by it, and *(ii)* the behavior of the global grammar can be recovered from the joint behavior of its components where all internal messages are consumed immediately.

PROPOSITION 3.2. *Let $G = G_1 \oplus \cdots \oplus G_p$ be a composition, Γ_i a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ the corresponding global configuration. Then*

1. $\Gamma \xRightarrow[M]{R/X} \Gamma' \implies \exists!\, \Gamma''$ *such that* $\Gamma \xRightarrow[M_G]{R/X} \Gamma''$

2. $\Gamma \xrightarrow[M_G]{e} \Gamma' \iff \Gamma \xRightarrow[M_G]{e} \Gamma'$

PROOF. The first statement follows from the fact that relation $\Gamma \xRightarrow[M]{e} \Gamma'$ is deterministic (M and Γ' are uniquely defined from Γ and e) and the application of a rule produces, directly or indirectly, a finite number of messages.

- If m is of the form $Y = s(t_1, \ldots, t_n)\langle y_1, \ldots, y_n\rangle$, then consuming m results in adding new equations to the local configuration that receives it, and generating a set of messages of the form $\bar{x} = x$ that can hence be consumed by the location that will receive them without generating new messages (case 3 below).

- If $m = (x = t)$, then consumption of the message results in production of new variables, and a new (finite) set of messages of the form $\bar{y} = y$ that can then be consumed by the location that send m without producing any new message.

- If $m = (y = x)$ then consuming the message results in adding an equation $y = x$ to the local configuration without generating any new message.

The second statement follows from the fact that the construction of a global configuration $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ amounts to consuming all the local messages between the components. *Q.E.D.*

COROLLARY 3.3. *Let $G = G_1 \oplus \cdots \oplus G_p$ be a composition where G is an autonomous guarded attribute grammar and Γ be a configuration of G. Then*

$$\Gamma[R/X\rangle\Gamma' \iff \Gamma \xRightarrow[\emptyset]{R/X} \Gamma'$$

A configuration is *stable* when all internal messages are consumed. The behavior of the global grammar is thus given as the restriction of the join behavior of the components to the set of stable configurations. This amounts to imposing that every event of the global configuration is immediately followed by the consumptions of the internal messages generated by the event. However if the various active workspaces are distributed at distant locations on an asynchronous architecture, one can never guarantee that no internal message remains in transit, or that some message is received but not yet consumed in some distant location. In order to ensure a correct distribution we therefore need a monotony property stating that *(i)* a locally enabled rule cannot become disbled by the arrival of a message and *(ii)* local actions and incomming messages can be swapped. We identify in the next section a class of guarded attribute grammars having this monotony property and which thus guarantees a safe implementation of distributed active workspaces.

4. PROPERTIES OF GAG

In this section we investigate some formal properties of guarded attribute grammars. We first turn our attention to *input-enabled* guarded attribute grammars to guarante that the application of a rule in an open node is a monotonous and confluent operation. This property is instrumental for the distribution of a GAG specification on an asynchronous architecture. We then consider *soundness*, a classical property of case management systems stating that any case introduced in the system can reach completion. We show that this property is undecidable. Nonetheless, soundness is preserved by hierarchical composition –a restrictive form of modular composition. This opens up the ability to obtain a large class of specifications that are sound by construction, if we start from basic specifications that are known to be sound by some ad-hoc arguments.

4.1 Distribution of a GAG

We say that rule R is **triggered** in node X if substitution σ_{in} –given in def. 2.9– is defined: Patterns p_i match the data d_i. As shown by the following example one can usually suspect a flaw in a specification when a triggered transition is not enabled due to the fact that the system of equations $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ is cyclic.

EXAMPLE 4.1. Let us consider the guarded attribute grammar given by the following rules:

$$\begin{array}{ll} P: & s_0(\,)\langle\,\rangle \to s_1(a(x))\langle x\rangle \quad s_2(x)\langle\,\rangle \\ Q: & s_1(y)\langle a(y)\rangle \quad \to \\ R: & s_2(a(z))\langle\,\rangle \quad \to \end{array}$$

Applying P in node X_0 of configuration $\Gamma_0 = \{X_0 = s_0(\,)\langle\,\rangle\}$ leads to configuration

$$\Gamma_1 = \{X_0 = P(X_1, X_2);\ X_1 = s_1(a(x))\langle x\rangle;\ X_2 = s_2(x)\langle\,\rangle\}$$

Rule Q is triggered in node X_1 with $\sigma_{in} = \{y = a(x)\}$ but the occur check fails because variable x occurs in $a(y)\sigma_{in} = a(a(x))$. Alternatively, we could drop the occur check and instead adapt the fixed point semantics for attribute evaluation defined in [7, 31] in order to cope with infinite data structures. More precisely, we let σ_{out} be defined as the least solution of system of equations $\{y_i = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ — assuming these equations are guarded, i.e., that there is no cycle of copy rules in the link graph of any accessible configuration. In that case the infinite tree a^ω is substituted to variable x and the unique maximal computation associated with the grammar is given by the infinite tree $P(Q, R^\omega)$. In Definition 2.11 we have chosen to restrict to finite data structures which seems a reasonable assumption in view of the nature of systems we want to model. The occur check is used to avoid recursive definitions of attribute values. The given example, whose most natural interpretation is given by fixed point computation, should in that respect be considered as ill-formed. And indeed this guarded attribute grammar is not *sound* —a notion presented in Section 4.2— because configuration Γ_1 is not closed (it still contains open nodes) but yet it is a terminal configuration that enables no rule. Hence it represents a case that can not be terminated.

End of Exple 4.1

The fact that triggered rules are not enabled can also impact the distributability of a grammar as shown by the following example.

Example 4.2. Let us consider the GAG with the following rules:

$$\begin{array}{ll} P: s(\,)\langle\,\rangle & \to s_1(x)\langle y\rangle \;\; s_2(y)\langle x\rangle \\ Q: s_1(z)\langle a(z)\rangle & \to \\ R: s_2(u)\langle a(u)\rangle & \to \end{array}$$

Rule P is enabled in configuration $\Gamma_0 = \{X_0 = s(\,)\langle\,\rangle\}$ with $\Gamma_0[P/X_0\rangle\Gamma_1$ where

$$\Gamma_1 = \{X_0 = P(X_1, X_2);\ X_1 = s_1(x)\langle y\rangle,\ X_2 = s_2(y)\langle x\rangle\}\,.$$

In configuration Γ_1 rules Q and R are enabled in nodes X_1 and X_2 respectively with $\Gamma_1[Q/X_1\rangle\Gamma_2$ where

$$\Gamma_2 = \{X_0 = P(X_1, X_2);\ X_1 = Q,\ X_2 = s_2(a(x))\langle x\rangle\}$$

and $\Gamma_1[R/X_2\rangle\Gamma_3$ where

$$\Gamma_3 = \{X_0 = P(X_1, X_2);\ X_1 = s_2(a(y))\langle y\rangle,\ X_2 = R\}$$

Now rule R is triggered but not enabled in node X_2 of configuration Γ_2 because of the cyclicity of $\{x = a(a(x))\}$. Similarly, rule Q is triggered but not enabled in node X_3 of configuration Γ_3. There is a conflict between the application of rules R and Q in configuration Γ_1. When the grammar is distributed in such a way that X_1 and X_2 have distinct locations, the specification is not implementable.

End of Exple 4.2

We first tackle the problem of safe distribution of a GAG specification on an asynchronous architecture by limiting ourselves to standalone systems. Hence to autonomous guarded attribute grammars. At the end of the section we show that this property can be verified in a modular fashion if the grammar is given as the composition of local (and thus non-autonomous) grammars.

Definition 4.3 (Accessible Configurations). *Let G be an autonmous guarded attribute grammar. A **case** $c = s(t_1, \ldots, t_n)\langle x_1, \ldots, x_m\rangle$ is a ground instanciation of service s, an axiom of the grammar, i.e., the values t_i of the inherited attributes are ground terms. It means that it is a service call which already contains all the information coming from the environment of the guarded attribute grammar. An **initial configuration** is any configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with a case c. An **accessible configuration** is any configuration accessible from an initial configuration.*

Substitution σ_{in}, given by pattern matching, is monotonous w.r.t. incoming information and thus it causes no problem for a distributed implementation of a model. However substitution σ_{out} is not monotonous since it may become undefined when information coming from a distant location makes the match of output attributes a cyclic set of equations, as illustrated by example 4.2.

Definition 4.4. *An autonomous guarded attribute grammar is **input-enabled** if every rule that is triggered in an accessible configuration is also enabled.*

If every form $s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle$ occurring in some reachable configuration is such that variables y_j do not appear in expressions d_i then by Remark 2.10 the guarded attribute grammar is input-enabled —moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ for every enabled rule. This property is clearly satisfied for guarded L-attributed grammars which consequently constitute a class of input-enabled garded attribute grammars.

Definition 4.5 (L-attributed Grammars). *A guarded attribute grammar is* left-attributed*, in short a LGAG, if any variable that is used in an inherited position in some form F of the right-hand side of a rule is either a variable defined in a pattern in the left-hand side of the rule or a variable occurring at a synthesized position in a form which appears at the left of F, i.e., inherited information flows from top-to-bottom and left-to-right between sibling nodes.*

We call the *substitution induced by a sequence* $\Gamma[*\rangle\Gamma'$ the corresponding composition of the various substitutions associated respectively with each of the individual steps in the sequence. If X is an open node in both Γ and Γ', i.e., no rules are applied at node X in the sequence, then we get $X = s(d_1\sigma, \ldots, d_n\sigma)\langle y_1, \ldots, y_m\rangle \in \Gamma'$ where

$$X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$$

and σ is the substitution induced by the sequence.

Proposition 4.6 (Monotony). *Let Γ be an accessible configuration of an input-enabled GAG, $X = s(d_1, \ldots, d_n)\langle y_1, \ldots, y_m\rangle \in \Gamma$ and σ be the substitution induced by some sequence starting from Γ. Then*

$$\Gamma[P/X\rangle\Gamma' \quad \text{implies} \quad \Gamma\sigma[P/X\rangle\Gamma'\sigma.$$

Proof. Direct consequence of Definition 2.3 due to the fact that

1. $\mathbf{match}(p, d\sigma) = \mathbf{match}(p, d)\sigma$, and

2. $\mathbf{mgu}(\{y_j = u_j\sigma \mid 1 \le j \le m\}) = \mathbf{mgu}(\{y_j = u_j \mid 1 \le j \le m\})\sigma$.

The former is trivial and the latter follows by induction on the length of the computation of the most general unifier –relation $\Rightarrow^*$ using rule (5) only. Note that the assumption that the guarded attribute grammar is input-enabled is crucial because in the general case it could happen that the set $\{y_j = u_j\sigma_{in} \mid 1 \le j \le m\}$ satisfies the occur check whereas the set $\{y_j = u_j(\sigma_{in}\sigma) \mid 1 \le j \le m\}$ does not satisfy the occur check. *Q.E.D.*

Proposition 4.6 is instrumental for the distributed implementation of guarded attribute grammars. Namely it states that new information coming from a distant asynchronous location refining the value of some input occurrences of variables of an enabled rule do not prevent the rule to apply. Thus a rule that is locally enabled can freely be applied regardless of information that might further refine the current partial configuration. It means that conflicts arise only from the existence of two distinct rules enabled in the same open node. Hence the only form of non-determinism corresponds to the decision of a stakeholder to apply one particular rule among those enabled in a configuration. This is expressed by the following confluence property.

COROLLARY 4.7. *Let Γ be an accessible configuration of an input enabled GAG. If $\Gamma[P/X\rangle\Gamma_1$ and $\Gamma[Q/Y\rangle\Gamma_2$ with $X \neq Y$ then $\Gamma_2[P/X\rangle\Gamma_3$ and $\Gamma_1[Q/Y\rangle\Gamma_3$ for some configuration Γ_3.*

Note that, by Corollary 4.7, the artifact contains a full history of the case in the sense that one can reconstruct from the artifact the complete sequence of applications of rules leading to the resolution of the case —up to the commutation of independent elements in the sequence.

REMARK 4.8. We might have considered a more symmetrical presentation in Definition 2.3 by allowing patterns for synthesized attributes in the right-hand sides of rules with the effect of creating forms in a configuration with patterns in their co-arguments. These patterns would express constraints on synthesized values. This extension could be acceptable if one sticks to purely centralized models. However, as soon as one wants to distribute the model on an asynchronous architecture, one cannot avoid such a constraint to be further refined due to a transformation occurring in a distant location. Then the monotony property (Proposition 4.6) is lost: A locally enabled rule can later be disabled when a constraint on a synthesized value gets a refined value. This is why we required synthesized attributes in the right-hand side of a rule to be given by plain variables in order to prohibit constraints on synthesized values.

End of Remark 4.8

It is difficult to verify input-enabledness as the whole set of accessible configurations is involved in this condition. Nevertheless one can find a sufficient condition for input enabledness, similar to the strong non-circularity of attribute grammars [9], that can be checked by a simple fixpoint computation.

DEFINITION 4.9. *Let s be a sort of a guarded attribute grammar with n inherited attributes and m synthesized attributes. We let $(j,i) \in SI(s)$ where $1 \le i \le n$ and $1 \le j \le m$ if there exists $X = s(d_1,\ldots,d_n)\langle y_1,\ldots,y_m\rangle \in \Gamma$ where Γ is an accessible configuration and $y_j \in d_i$. If R is a rule with left-hand side $s(p_1,\ldots,p_n)\langle u_1,\ldots,u_m\rangle$ we let $(i,j) \in IS(R)$ if there exists a variable $x \in \mathrm{var}(R)$ such that $x \in \mathrm{var}(d_i) \cap \mathrm{var}(u_j)$. The guarded attribute grammar G is said to be **acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $G(s,R) = SI(s) \cup IS(R)$ is acyclic.*

PROPOSITION 4.10. *An acyclic guarded attribute grammar is input-enabled.*

PROOF. Suppose R is triggered in node X with substitution σ_{in} such that $y_j \in u_i\sigma_{in}$ then $(i,j) \in G(s,R)$. Then the fact that occur check fails for the set $\{y_j \mid 1 \le j \le m\}$ entails that one can find a cycle in $G(s,R)$. *Q.E.D.*

Relation $SI(s)$ still takes into account the whole set of accessible configurations. The following definition provides an overapproximation of this relation given by a fixpoint computation.

DEFINITION 4.11. *The **graph of local dependencies** of a rule $R : F_0 \to F_1 \cdots F_\ell$ is the directed graph $GLD(R)$ that records the data dependencies between the occurrences of attributes given by the semantics rules. We designate the occurrences of attributes of R as follows: We let $k(i)$ (respectively $k\langle j\rangle$) denote the occurrence of the i^{th} inherited attribute (resp. the j^{th} synthesized attribute) in F_k. If s is a sort with n inherited attributes and m synthesized attributes we define the relations $\overline{IS(s)}$ and $\overline{SI(s)}$ over $[1,n]\times[1,m]$ and $[1,m]\times[1,n]$ respectively as the least relations such that:*

1. *For every rule $R : F_0 \to F_1 \cdots F_\ell$ where form F_i is of sort s_i and for every $k \in [1,\ell]$*

$$\left\{(j,i) \;\middle|\; (k\langle j\rangle, k(i)) \in GLD(R)^k\right\} \subseteq \overline{SI(s_k)}$$

where graph $GLD(R)^k$ is given as the transitive closure of

$$GLD(R) \cup \left\{(0\langle j\rangle, 0(i)) \;\middle|\; (j,i) \in \overline{SI(s_0)}\right\} \cup \left\{(k'(i), k'\langle j\rangle) \;\middle|\; k' \in [1,\ell],\ k' \neq k,\ (i,j) \in \overline{IS(s_{k'})}\right\}$$

2. *For every rule $R : F_0 \to F_1 \cdots F_\ell$ where form F_i is of sort s_i*

$$\left\{(i,j) \;\middle|\; (0(i), 0\langle j\rangle) \in GLD(R)^0\right\} \subseteq \overline{IS(s_0)}$$

where graph $GLD(R)^0$ is given as the transitive closure of

$$GLD(R) \cup \left\{(k(i), k\langle j\rangle) \;\middle|\; k \in [1,\ell],\ (i,j) \in \overline{IS(s_k)}\right\}$$

*The guarded attribute grammar G is said to be **strongly-acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $\overline{G(s,R)} = \overline{SI(s)} \cup IS(R)$ is acyclic.*

PROPOSITION 4.12. *A strongly-acyclic guarded attribute grammar is acyclic and hence input-enabled.*

PROOF. The proof is analog to the proof that a strongly non-circular attribute grammar is non-circular and it goes as follows. We let $(i,j) \in IS(s)$ when $\mathrm{var}(d_i\sigma) \cap \mathrm{var}(y_j\sigma) \neq \emptyset$ for some form $F = s(d_1,\ldots,d_n)\langle y_1,\ldots,y_m\rangle$ of sort s and where σ is the substitution induced by a firing sequence starting from configuration $\{X = F\}$. Then we show by induction on the length of the firing sequence leading to the accessible configuration that $IS(s) \subseteq \overline{IS(s)}$ and $SI(s) \subseteq \overline{SI(s)}$. *Q.E.D.*

Note that the following two inclusions are strict

strongly-acyclic GAG $\subsetneq$ acyclic GAG $\subsetneq$ input enabled GAG

Indeed the reader may easily check that the guarded attribute grammar with rules

$$\begin{cases} A(x)\langle z\rangle \to B(a(x,y))\langle y,z\rangle \\ B(a(x,y))\langle x,y\rangle \to \end{cases}$$

is cyclic and input-enabled whereas guarded attribute grammar with rules

$$\begin{cases} A(x)\langle z\rangle \to B(y,x)\langle z,y\rangle \\ A(x)\langle z\rangle \to B(x,y)\langle y,z\rangle \\ B(x,y)\langle x,y\rangle \to \end{cases}$$

is acyclic but not strongly-acyclic. Attribute grammars arising from real situations are almost always strongly non-circular so that this assumption is not really restrictive. Similarly we are confident that most of the guarded attribute grammars that we shall use in practise will be input-enabled and that most of the input-enabled guarded attribute grammars are in fact strongly-acyclic. Thus most of the specifications are distributable and in most cases, this can be proved by checking strong non-circularity.

Let us conclude this section by addressing the modularity of strong-acyclicity. This property (see Def. 4.11) however was defined for autonomous guarded attribute grammars viewed as standalone applications. Here, the initial configuration is associated with a case $c = s(t_1,\ldots,t_n)\langle y_1,\ldots,y_m\rangle$ introduced by the external environment. And the information transmitted to the inherited attributes are given by ground terms. Even though one can imagine that these values are introduced gradually they do not depend on the values that will be returned to the subscribing variables $y_1,\ldots,y_m$. It is indeed reasonable to assume that when an user enters a new case in the system she instanciates the inherited information and then waits for the returned values. Things go differently if the autonomous guarded attribute grammar is not a standalone application but a component in a larger specification. In that case, a call to the service provided by this component is of the form $s(t_1,\ldots,t_n)\langle y_1,\ldots,y_m\rangle$ where the values transmitted to the inherited attributes may depend (directly or indirectly) on the subscribing variables. Definition 4.11 should be amended to incorporate these dependencies and strong-acyclicity can be lost. Therefore a component which is strongly-acyclic when used as a standalone application might lose this property when it appears as an individual component of a larger specification. Thus if the component is already implemented as a collection of subcomponents distributed on an asynchronous architecture, the correctness of this distribution can be lost if the component takes part in a larger system.

To avoid this pitfall we follow a standard contract based approach, where each component can be developped independently as long as it conforms to contraints given by assume/guarantee conditions. Assuming some properties of the environment, this approach allows to preserve properties of assembled components. In our case, we show that strong-acyclicity is preserved by composition.

DEFINITION 4.13. *Let s be a sort with n inherited attributes and m synthesized attributes. We let $\mathbf{IS}(s) = [1,n]\times[1,m]$ and $\mathbf{SI}(s) = [1,m]\times[1,n]$ denote the set of (potential) dependencies between inherited and synthesized attributes of s. Let G be a guarded attribute grammar with axioms $s_1,\ldots,s_k$ and terminal symbols $s'_1,\ldots,s'_{k'}$. An **assume/guarantee condition** for G is a pair $(a,g) \in \mathbf{AG}(G)$ with $a \in \mathbf{SI}(s_1)\times\cdots\times\mathbf{SI}(s_k)\times\mathbf{IS}(s'_1)\times\cdots\times\mathbf{IS}(s'_{k'})$ and $g \in \mathbf{IS}(s_1)\times\cdots\times\mathbf{IS}(s_k)\times\mathbf{SI}(s'_1)\times\cdots\times\mathbf{SI}(s'_{k'})$. Equivalently it is given by the data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ for $s \in \mathbf{axioms}(G)\cup T$. The guarded attribute grammar G is strongly-acyclic w.r.t. assume/guarantee condition (a,g) if the modified fixed-point computation of Def. 4.11, where constraints $SI(s) \subseteq \overline{SI(s)}$ for $s \in \mathbf{axioms}(G)$ and $IS(s) \subseteq \overline{IS(s)}$ for $s \in T$ are added, allows to conclude strong-acyclicity with $\overline{IS(s)} \subseteq IS(s)$ for $s \in \mathbf{axioms}(G)$ and $\overline{SI(s)} \subseteq SI(s)$ for $s \in T$.*

The data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ give an over-approximation of the attribute dependencies, the so-called 'potential' dependencies, for the axioms and the terminal symbols. They define a *contract* of the guarded attribute grammar. This contract splits into *assumptions* about its environment –SI dependencies for the axioms and IS dependencies for the terminal symbols– and *guarantees* offered in return to the environment –IS dependencies for the axioms and SI dependencies for the terminal symbols. Thus strongly-acyclicity of a guarded attribute grammar G w.r.t. assume/guarantee condition (a,g) means that when the environment satisfies the assume condition, grammar G is strongly-acyclic and satisfies the guarantee condition.

The following result states the modularity of strong-acyclicity.

PROPOSITION 4.14. *Let $G = G_0 \oplus \cdots \oplus G_p$ be a composition of guarded attribute grammars. Let $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ be assumptions on the (potential) dependencies between attributes where sort s ranges over the set of axioms and terminal symbols of the components G_i –thus containing also the axioms and terminal symbols of global grammar G. These constraints restrict to assume/guarantee conditions $(a_i,g_i) \in \mathbf{AG}(G_i)$ for every local grammar and for the global grammar as well: $(a,g) \in \mathbf{AG}(G)$. Then G is strongly-acyclic w.r.t. (a,g) when each local grammar G_i is strongly-acyclic w.r.t. (a_i,g_i).*

PROOF. The fact that the fixed-point computation for the global grammar can be computed componentwise follows from the fact that for each local grammar no rule apply

locally to a terminal symbol s and consequently rule 3 in Def. 4.11 never applies for s and the value $\overline{SI(s)}$ is left unmodified during the fixpoint computation, it keeps its initial value $SI(s)$. Similarly, rule 2 in Def. 4.11 never applies for an axiom s and $\overline{IS(s)}$ keeps its initial value $IS(s)$.

Q.E.D.

Conversely if the global grammar is strongly-acyclic w.r.t. some assume/guarantee condition (a, g) then the values of $\overline{SI(s)}$ and $\overline{IS(s)}$ produced at the the end of the fixpoint computation allows to complement the assume/guarantee conditions with respect to which the local grammars are strongly-acyclic. The issue is how to guess some correct assume/guarantee conditions in the first place. One can imagine that some knowledge about the problem at hand can help to derive the potential attribute dependencies, and that we can use them to type the components for their future reuse in larger specifications. In many cases however there is no such dependencies and the assume/guarantee conditions are given by empty relations.

DEFINITION 4.15. *A composition $G = G_0 \oplus \cdots \oplus G_p$ of guarded attribute grammars is **distributable** if each local grammar (and hence also the global grammar) is strongly-acyclic w.r.t. the empty assume/guarantee condition.*

This condition might seems rather restrictive but it is not. Indeed $(i, j) \notin IS(s)$ (similarly for $(i, j) \notin SI(s)$) does not mean that the j^{th} synthesized attribute does not depend on the value received by the i^{th} inherited attribute –the inherited value influences the behaviour of the component and hence has an impact on the values that will be returned in synthesized attributes. It rather says that one should not return in the value of a synthesized attribute some data directly extracted from the value of inherited attributes, which is the most common situation. The emptiness of assume/guarantee gives us a criterion for a distributable decomposition of a guarded attribute grammar: It indicates the places where a specification can safely be split into smaller pieces. We shall denote a distributable composition as:

$$\begin{array}{ll} G = \langle G_1, \ldots, G_p \rangle & \\ \textbf{where} & G_1 :: \%\text{definition of } G_1 \\ & \vdots \\ & G_p :: \%\text{definition of } G_p \end{array}$$

4.2 Soundness

A specification is *sound* if every case can reach completion no matter how its execution started. Recall from Def. 4.3 that a case $c = s_0(t_1, \ldots, t_n)\langle x_1, \ldots, x_m \rangle$ is a ground instanciation of service s_0, an axiom of the grammar. And, an accessible configuration is any configuration accessible from a configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with a case c (an initial configuration).

DEFINITION 4.16. *A configuration is **closed** if it contains only closed nodes. An autonomous guarded attribute grammar is **sound** if a closed configuration is accessible from any accessible configuration.*

We consider the finite sequences $(\Gamma_i)_{0<i\leq n}$ and the infinite sequences $(\Gamma_i)_{0<i<\omega}$ of accessible configurations such that $\Gamma_i[\rangle\Gamma_{i+1}$. A finite and maximal sequence is said to be **terminal**. Hence a terminal sequence leads to a configuration that enables no rule. Soundness can then be rephrased by the two following conditions.

1. Every terminal sequence leads to a closed configuration.
2. Every configuration on an infinite sequence also belongs to some terminal sequence.

Soundness can unfortunately be proved indecidable by a simple encoding of Minsky machines.

PROPOSITION 4.17. *Soundness of guarded attribute grammar is undecidable.*

PROOF. We consider the following presentation of the Minsky machines. We have two registers r_1 and r_2 holding integer values. Integers are encoded with the constant **zero** and the unary operator **succ**. The machine is given by a finite list of instructions $instr_i$ for $i = 1, \ldots, N$ of one of the three following forms

1. **INC(r, i):** increment register r and go to instruction i.
2. **JZDEC(r, i, j):** if the value of register r is 0 then go to instruction i else decrement the value of the register and go to instruction j.
3. **HALT:** terminate.

We associate a Minsky machine with a guarded attribute grammar whose sorts corresponds bijectively its instructions, $S = \{s_1, \ldots, s_N\}$, with the following encoding of the program instructions by rules:

1. If $instr_k = \text{INC}(r_1, i)$ then add rule
$$\text{Inc}(k, 1, i) : s_k(x, y) \to s_i(\mathbf{succ}(x), y)$$
2. If $instr_k = \text{INC}(r_2, i)$ then add rule
$$\text{Inc}(k, 2, i) : s_k(x, y) \to s_i(x, \mathbf{succ}(y))$$
3. If $instr_k = \text{JZDEC}(r_1, i, j)$ then add the rules
$$\begin{array}{ll} \text{Jz}(k, 1, i) & : s_k(\mathbf{zero}, y) \to s_i(\mathbf{zero}, y) \\ \text{Dec}(k, 1, j) & : s_k(\mathbf{succ}(x), y) \to s_j(x, y) \end{array}$$
4. If $instr_k = \text{JZDEC}(r_2, i, j)$ then add the rules
$$\begin{array}{ll} \text{Jz}(k, 2, i) & : s_k(x, \mathbf{zero}) \to s_i(x, \mathbf{zero}) \\ \text{Dec}(k, 2, j) & : s_k(x, \mathbf{succ}(y)) \to s_j(x, y) \end{array}$$
5. If $instr_k = \text{HALT}$ then add rule
$$\text{Halt}(k) : s_k(x, y) \to$$

Since there is a unique maximal firing sequence from the initial configuration $\Gamma_0 = \{X_0 = s_1(\mathbf{zero}, \mathbf{zero})\}$ the corresponding guarded attribute grammar is sound if and only if the computation of the corresponding Minsky machine terminates. *Q.E.D.*

This result is not surprising as most of non-trivial properties of an expressive enough formalism are indeed undecidable. The above encoding uses very simple features of the model: All guards are of depth at most one, the system is deterministic, and there are only inherited attributes (and in fact only two of them)! This leaves no hope to find syntactic restrictions to characterize an effective subclass of sound specifications.

Even though this problem is undecidable, soundness can still be proven for a given specification using ad-hoc arguments. Furthermore we show now that a restricted form of composition of GAG, called *hierarchical composition* preserves soundness. This result allows the construction of a large class of specifications which are sound by construction.

DEFINITION 4.18 (HIERARCHICAL COMPOSITION). *A composition $G = G_0 \oplus \cdots \oplus G_n$ is **hiearchical** if each GAG G_i has a unique axiom s_i, the local grammars $C_i = G_i$ for $1 \leq i \leq n$ are autonomous, and the terminal symbols of $K = G_0$ are $\{s_1, \ldots, s_n\}$.*

We interpret K as a *connector* that defines the possible orchestrations of the services associated with components $C_1, \ldots, C_n$ and denote $G = K(C_1, \ldots, C_n)$ such a composition. In this situation, depicted in Fig. 12, the connector provides a global service s_0 through an orchestration of the components. For that purpose the connector can make service calls to the services $s_1, \ldots, s_n$ delivered by the components. The components are autonomous and therefore can not call their respective services. They can however communicate with each other information via attributes but only indirectly using the connector. Thus orchestration between the various components is fully encoded into the connector specification. The resulting composition, $C = K(C_1, \cdots, C_n)$, is also an autonomous guarded attribute grammar and thus one can iterate this construction to obtain more complex hiarchical decompositions.

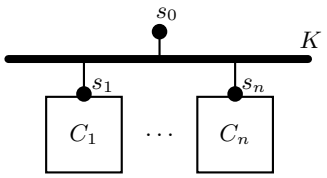


Figure 12: A hierarchical composition of GAGs

Soundness was defined in Def. 4.16 for autonomous guarded attribute grammars. We adapt this definition for a connector as follows.

DEFINITION 4.19. *A guarded attribute grammar is a **connector** if it satisfies the following two conditions:*

1. ***Weak-Soundness.*** *For every accessible configuration Γ there exists a configuration accessible from Γ all of whose open nodes have sorts that are terminal symbols.*

2. ***Independence w.r.t. Components.*** *For any accessible configuration Γ, open node X, and substitution σ for variables subscribing to external services (i.e., occurring in a synthesized position in an open node of Γ whose sort is a terminal symbol), and for any rule R one has $\Gamma[R/X\rangle \Leftrightarrow \Gamma\sigma[R/X\rangle$, which means that a pattern of a rule never tests values produced by an external service.*

Intuitively Independence w.r.t. Components means that components can exchange information through the connector but this information has no impact on the choices of rules to apply within the connector.

PROPOSITION 4.20. *Let $C = K(C_1, \ldots, C_n)$ be a hierarchical composition where K is a connector and the components $C_1, \ldots, C_n$ are sound. If the global grammar is input-enabled —for instance if the composition is distributable— then it is also a sound (autonomous) guarded attribute grammar.*

PROOF. Let Γ be an accessible configuration of C. By Independence w.r.t. Components there exists sequences of rule applications $\Gamma_0[*\rangle\Gamma_1[*\rangle\Gamma$ where rules in the first sequence belong to the connector, and in the second sequence belong to the components. By Weak-Soundness, as Γ_1 is a configuration of the connector, one can find a sequence of rule applications in the connector $\Gamma_1[*\rangle\Gamma_2$ such that the sort of an open node in Γ_2 is a terminal symbol of the connector (i.e., lies in $I(K)$). By Confluence (which follows from Input-Enabledness) there exists a configuration Γ_3 which is accessible both from Γ and from Γ_2. The sorts of open nodes of Γ_3 are found in the components. By Soundness of the components and Monotony (which also follows from Input-Enabledness) $\Gamma_3[*\rangle\Gamma_4$ where Γ_4 is a closed configuration. *Q.E.D.*

5. TOWARDS A LANGUAGE FOR THE SPECIFICATION OF GAG

In this section we introduce some syntax elements to outline a specification language for guarded attribute grammars that is expressive enough to describe realistic applications. Our purpose is not to fully design such a specification language. This would require more thorough investigations and, in particular, the implementation of some typing mechanism for the manipulated values. We only intend to introduce some syntatic sugar and constructs which allow to describe large and complex specifications in a more concise and friendlier way. This syntax is used in Section 6 where a case study is presented.

First, we introduce in Section 5.1 a functional notation for business rules, inspired from monadic programming in Haskell. So far, a guarded attribute grammar was presented as a task rewriting system, a convenient formalism for formal

manipulations. However, rewriting systems are not necessarily perceived as a handy programming notation despite their similarity with logic programming. In Section 5.2 we give the opportunity to write generic rules, namely rules that contain parameters whose instanciations can generate a potentially large set of similar rules. This is particularly useful to formalize the notion of role: When several stakeholders play a similar role they can use the same generic local grammar instantiated with their respective identities to distinguish them from one another. Section 5.3 introduces a feature that allows the designer to extend the formalism by adding combinators, a technique that can be used to customize the notations in order to derive domain specific languages adapted to the particular user needs.

5.1 A Functional Notation

In order to ease the writing of rules we introduce a syntax inspired from monadic computations in Haskell. More precisely, we restate rule

$$\begin{aligned} &sort(p_1,\ldots,p_n)\langle u_1,\ldots,u_m\rangle \rightarrow \\ &\qquad sort_1(t_1^{(1)},\ldots,t_{n_1}^{(1)})\langle y_1^{(1)},\ldots,y_{m_1}^{(1)}\rangle \\ &\qquad \cdots \\ &\qquad sort_k(t_1^{(k)},\ldots,t_{n_k}^{(k)})\langle y_1^{(k)},\ldots,y_{m_k}^{(k)}\rangle \end{aligned}$$

as

$$\begin{aligned} &sort(p_1,\ldots,p_n) \;= \\ &\quad \textbf{do} \quad (y_1^{(1)},\ldots,y_{m_1}^{(1)}) \leftarrow sort_1(t_1^{(1)},\ldots,t_{n_1}^{(1)}) \\ &\qquad \cdots \\ &\qquad (y_1^{(k)},\ldots,y_{m_k}^{(k)}) \leftarrow sort_k(t_1^{(k)},\ldots,t_{n_k}^{(k)}) \\ &\qquad \textbf{return}\ (u_1,\ldots,u_m) \end{aligned}$$

This functional presentation stresses out the operational purpose of business rules: Each task has an input –inherited attributes– seen as parameters and an output –synthesized attributes– seen as returned values. This notation however can confuse Haskell programmers for two reasons.

First, recall from Definition 2.3 that an input occurrence of a variable is either a variable occurring in a pattern p_i or a variable occurring as a subscription in the right-hand side of the rule or, in this alternative presentation, in the left-hand side of a *generator*

$$(y_1^{(j)},\ldots,y_{m_j}^{(j)}) \leftarrow sort_j(t_1^{(j)},\ldots,t_{n_j}^{(j)})$$

Note that, using this **do** notation, a guarded attribute grammar is left-attributed (Def. 4.5) precisely when every variable is defined before used: each output occurrence of a variable is preceded by its corresponding input occurrence. For guarded attribute grammar which are not left-attributed we thus find some variables which are used before being defined. This is incompatible with monadic programming in Haskell where the scope of a variable occurring in the left-hand side of a generator is the part of the **do** expression that follows the generator, including the return statement.

Second, a Haskell monadic expression is evaluated in *pull* mode: If the output returned by the **do** expression does not use the values of the variables defined by a given generator, then this generator is not evaluated at all. By contrast, a rule of a guarded attribute grammar is evaluated in *push* mode: When rule is applied, we create one open node for every generator. Then users can continue to develop these nodes with the effect of gradually refining the returned values.

Example 5.1. Consider the GAG of Example 2.2:

$$\begin{aligned} &\text{Root} \;:\; root()\langle x\rangle \rightarrow bin(\text{Nil})\langle x\rangle \\ &\text{Fork} \;:\; bin(x)\langle y\rangle \rightarrow bin(z)\langle y\rangle\ bin(x)\langle z\rangle \\ &\text{Leaf}_a \;:\; bin(x)\langle \text{Cons}_a(x)\rangle \rightarrow \end{aligned}$$

Its syntactical translation into the functional notation is the following:

$$\begin{aligned} &\text{Root}: \quad \text{root}() \;=\; \textbf{do} \quad (x) \leftarrow bin(\text{Nil}) \\ &\qquad\qquad\qquad\qquad\quad\ \ \textbf{return}\ (x) \\ &\text{Fork}: \quad \text{bin}(x) \;=\; \textbf{do} \quad (z) \leftarrow bin(x) \\ &\qquad\qquad\qquad\qquad\quad\ \ (y) \leftarrow bin(z) \\ &\qquad\qquad\qquad\qquad\quad\ \ \textbf{return}\ (y) \\ &\text{Leaf}_a \quad \text{bin}(x) \;=\; \textbf{do} \quad \textbf{return}\ (\text{Cons}_a(x)) \end{aligned}$$

which we write more simply as

$$\begin{aligned} &\text{Root}: \quad \text{root}() \;=\; bin(\text{Nil}) \\ &\text{Fork}: \quad \text{bin}(x) \;=\; \textbf{do} \quad (z) \leftarrow bin(x) \\ &\qquad\qquad\qquad\qquad\quad\ \ (y) \leftarrow bin(z) \\ &\qquad\qquad\qquad\qquad\quad\ \ \textbf{return}\ (y) \\ &\text{Leaf}_a \quad \text{bin}(x) \;=\; \textbf{return}\ (\text{Cons}_a(x)) \end{aligned}$$

using the simplification rules given below.

End of Exple 5.1

(**SR**$_1$) When the returned value is the result of the last generator, one replaces these two instructions by the last call, e.g.:

$$\begin{aligned} &\textbf{do} \quad (y) \leftarrow bin\ (\text{Nil}) \\ &\qquad\ \ \textbf{return}\ (y) \end{aligned} \quad \Leftrightarrow \quad \textbf{do}\ bin\ (\text{Nil})$$

(**SR**$_2$) When the **do** sequence is reduced to a unique item –either a call or a **return** statement– one omits the **do** instruction, e.g.:

$$\begin{aligned} &\textbf{do}\ bin(\text{Nil}) && \Leftrightarrow && bin(\text{Nil}) \\ &\textbf{do}\ \textbf{return}\ (\text{Cons}_a(x)) && \Leftrightarrow && \textbf{return}\ (\text{Cons}_a(x)) \end{aligned}$$

5.2 Parametric Rules

It may happen that several components of a composition $G = G_1 \oplus \cdots \oplus G_k$ are associated with stakeholders that play the same *role* in the system and consequently use the same set of rules. To illustrate this situation let us consider the editorial process of a scholarly journal. The Editor of the journal has to make an editorial decision about a submitted paper by resorting to scientific evaluations produced by independent reviewers. The first local grammar consists of rules governing the activities of the Editor. The other local grammars describe the activities of the various reviewers. Suppose that the role of a reviewer was previously described by a guarded attribute grammar 'evaluate' with an homonymous axiom corresponding to the service "making a scientific evaluation of a paper" *–It is actually convenient to name a GAG by its axiom when this axiom is unique.* All the components associated with reviewers are thus given by this specification. However the components must have disjoint sets of non-terminal symbols so that the distribution schema described in Section 3 works properly.

For that purpose, each actual reviewer is attached to a *disjoint* copy of grammar 'evaluate'. Technically, a parameter *reviewer* is added to each of the non-terminal symbols of grammar 'evaluate'. Each component is then obtained by instanciating the parameter by an identifier of the corresponding reviewer. For instance evaluate[*reviewer*] is a generic sort and an actual sort is evaluate[Paul] where Paul is a reviewer. This allows to distinguish activities evaluate[Paul] and evaluate[Ann] that correspond to sending a paper for evaluation to Paul and to Ann respectively. We write –using a hierarchical form of composition:

editorial_decision(evaluate[*reviewer*]) **where**
reviewer = Paul | Ann | ...
editorial_decision :: %Grammar editorial_decision
evaluate :: %Description of the grammar evaluate

This contruct promotes an ordinary GAG to a generic one. At the same time, it alleviates the notations by locating the use of parameters: The sort evaluate[*reviewer*] appears in grammar 'editorial_decision' but grammar 'evaluate' simply uses sort evaluate. Such a construction can be iterated. For instance if the journal has several editors, one may write

editorial_process(editorial_decision[*editor*]) **where**
editor = Mary | Frank | ...
editorial_process :: %Grammar editorial_process
editorial_decision ::
  editorial_decision(evaluate[*reviewer*]) **where**
  reviewer = Paul | Ann | ...
  editorial_decision :: %...
  evaluate :: %...

The convention to name a GAG by its axiom entails some overloading of notations because in a hierarchical composition, the axiom of a global grammar coincides with the axiom of the connector. Nevertheless this overloading creates no confusion and it simplifies the notations. The above description makes it clear that the editorial process relies on editorial decisions made by editors and that such a decision depends on evaluations made by reviewers.

Parameters are instanciated by the connector. For instance a rule in grammar 'editorial_process' states that the Editor takes a decision based on evaluation reports produced by two referees. This rule can be written as:

Evaluate_Submission[$reviewer1, reviewer2$]
where not($reviewer1 = reviewer2$) :
submission($article$) =
  do $report1 \leftarrow$ evaluate[$reviewer1$]
    $report2 \leftarrow$ evaluate[$reviewer2$]
    decide($report1, report2$)

Evaluate_Submission[$reviewer1, reviewer2$] is a *generic rule*, the corresponding actual rules are obtained by choosing values for the parameters that conform to the condition given in the **where** clause. Thus it defines as many rules as pairs of distinct reviewers. We'd rather write the above rule on the form

Evaluate_Submission :
submission($article$) =
  input ($reviewer1, reviewer2$)
  where not($reviewer1 = reviewer2$)
  do $report1 \leftarrow$ evaluate[$reviewer1$]
    $report2 \leftarrow$ evaluate[$reviewer2$]
    decide($report1, report2$)

The **input** clause serves to highlight those parameters of the rule that are instanciated by the user when she applies this rule at an open node associated with task submission($article$). The corresponding instance of the generic rule, for instance

Evaluate_Submission[Paul, Ann]

which is the actual rule selected by the user, will subsequently label the node. In this manner the information about the selection of the reviewers is stored in the artifact. Parameters in the **input** clause enable user to input any kind of data and not solely instances of roles. For instance one may find the following rule in the specification of the reviewer:

Accept :
evaluate($article$) =
  input ($msg :: String$)
  do $report \leftarrow$ review($article$)
    return (Yes($msg, report$))

With this rule the reviewer informs the Editor that he accepts to review the paper. The returned value is formed with the Yes constructor, witnessing acceptance, with two arguments: A complementary (optional) message and a link to the report that the reviewer commits himself to subsequently produce. In this way the specification generates an infinite set of actual rules since there exists an infinite number of potential messages –including the empty one. In practice however the parameters will correspond either to a specific role in the system –whose instantiations are finite in number– or some kind of data –a message, a report, a decision, etc.– whose values should be kept in the artifact but has no impact on the subsequent behavior of the system. Therefore it will be possible to abstract the parameter values to end up with a finite guarded attribute grammar with the same behavior.

If we expand the global grammar 'editorial_decision' the rules of grammar 'evaluate' are promoted to generic rules. In particular the above rule gives rise to the following generic rule

Accept :
evaluate[$reviewer$]($article$) =
  input ($msg :: String$)
  do $report \leftarrow$ review[$reviewer$]($article$)
    return (Yes($msg, report$))

Note that by contrast to parameters that appear in an **input** clause, the parameters in the left-hand side of the rule do not correspond to user choices, simply because they are already instanciated in the current configuration. For instance, in an open node associated with task evaluate[Paul]($article$), which belongs to the active workspace of Paul, only the instance of the rule associated with Paul can apply: Mary has

no possibility to accept to review a paper that was not send to her.

All parameters of rules appear either in an **input** clause or in its left-hand side. Therefore they are always left implicite in the name of the rule.

We add the following rule to the simplification rules introduced in Sect. 5.1:

($\mathbf{SR}_3$) We omit the **do** statement if it merely returns the value(s) introduced by the preceding **input** clause:

$$\begin{array}{l}\textbf{input}\,(x)\\ \textbf{return}\,(x)\end{array} \quad \Leftrightarrow \quad \textbf{input}\,(x)$$

5.3 Combinators

Since we use a variant of attribute grammars and a notation inspired from monadic computations in Haskell the question naturally arises whether one can implement our specification language as a set of monadic combinators, in the line of the Parsec library [19] for functional parsers. In the present state of things it is not at all certain that such a goal is easily achievable or even feasible. Nonetheless, some useful combinators can be introduced to tailor the specification language towards more specific application domains.

For instance, one can introduce an iteration schema given by combinator **many** defined as follows. If s is a sort with inherited and synthesized attributes of respective types a and b, **many** s stands for a new sort whose inherited and synthesized attributes are lists of elements of type a and b respectively, associated with rules

$$\begin{array}{lll}\textbf{many}\ s\ (\mathrm{Cons}(\mathrm{head},\mathrm{tail})) \ = \ \textbf{do} & \mathrm{head}' \leftarrow s(\mathrm{head}) \\ & \mathrm{tail}' \leftarrow \textbf{many}\ s\ (\mathrm{tail}) \\ & \textbf{return}\ (\mathrm{Cons}(\mathrm{head}',\mathrm{tail}')) \\ \textbf{many}\ s\ (\mathrm{Nil}) \ = \ \textbf{return}\ (\mathrm{Nil}) & \end{array}$$

This is only syntactic sugar: a GAG uses a set of sorts with no *a priori* structure, but one can equip the set of sorts with a set of combinators together with a type system to constraint their usage –for instance **many** $s :: a^* \rightsquigarrow b^*$ when $s :: a \rightsquigarrow b$– and with rules which conform to this type system.

The following example, adapted from Example 2.18, uses the above combinator to describe two recurring tasks that communicate through lazy lists:

$$\begin{array}{lll}\langle s, s'\rangle :: () \rightsquigarrow () & & \\ \textbf{when} \quad s :: a \rightsquigarrow b & & \\ \qquad\quad s' :: b \rightsquigarrow a & & \\ \langle s, s'\rangle \ = \ \textbf{input}\ (x)\ \textbf{where}\ x :: a & & \\ \qquad \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(x, xs)) & & \\ \qquad\quad xs \leftarrow \textbf{many}\ s'\ (ys) & & \\ \qquad\quad \textbf{return}\ () & & \end{array}$$

The corresponding compound process uses no information from and returns no information to its surrounding environment. We can improve on this example by replacing this combinator by a (potentially infinite) set of combinators using functions as extra parameters:

$$\begin{array}{lll}\langle s, s'\rangle\ in\ out :: c \rightsquigarrow d & & \\ \textbf{when} \quad s :: a \rightsquigarrow b & & \\ \qquad\quad s' :: b \rightsquigarrow a & & \\ \qquad\quad in :: c \rightarrow a & & \\ \qquad\quad out :: a^* \times b^* \rightarrow d & & \\ \langle s, s'\rangle\ in\ out \ = \ \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(in\ x, xs)) & & \\ \qquad\qquad\quad xs \leftarrow \textbf{many}\ s'\ (ys) & & \\ \qquad\qquad\quad \textbf{return}\ (out\ (xs, ys)) & & \end{array}$$

This combinator has two parameters given by variables *in* and *out* and thus it actually generate a potentially infinite set of rules according to the actual functions used to instantiate the parameters. Still, if we assume that all such parametric combinators are totally instantiated in a global GAG specification then we guarantee that we end up with a finite specification.

For instance we may define the derived combinator:

$$\begin{array}{ll}[s, s'] = \langle s, s'\rangle id\ zip :: a \rightsquigarrow (a \times b)^* \\ \textbf{when} \quad s :: a \rightsquigarrow b \\ \qquad\quad s' :: b \rightsquigarrow a\end{array}$$

where $id :: a \rightarrow a$ is the identity function and $zip :: a^* \times b^* \rightarrow (a \times b)^*$ is the function given by:

$$\begin{array}{l} zip\ \mathrm{Nil}\ ys \ = \ \mathrm{Nil} \\ zip\ xs\ \mathrm{Nil} \ = \ \mathrm{Nil} \\ zip\ \mathrm{Cons}(x, xs)\ \mathrm{Cons}(y, ys) \ = \ \mathrm{Cons}((x, y), zip\ xs\ ys)\end{array}$$

This combinator can equivalently be specified as

$$\begin{array}{ll}[s, s'] :: a \rightsquigarrow (a \times b)^* \\ \textbf{when} \quad s :: a \rightsquigarrow b \\ \qquad\quad s' :: b \rightsquigarrow a \\ [s, s'] \ = \ \textbf{do} \quad ys \leftarrow \textbf{many}\ s\ (\mathrm{Cons}(x, xs)) \\ \qquad\quad xs \leftarrow \textbf{many}\ s'\ (ys) \\ \qquad\quad \textbf{return}\ (zip\ xs\ ys)\end{array}$$

However, this specification is not a GAG because the semantic rules are no longer given by plain terms but they also include some basic functions (like *zip*). One can impose that functions that instantiate the parameters of combinators are basic functions for list and tuples manipulations. These functions are, in any case, necessary to describe the various plumbing operations between the parametric combinators and their context of use. Since these functions are lazily evaluated it does not really affect the operational semantic of GAG.

6. A DISEASE SURVEILLANCE SYSTEM

In this section, we illustrate the model of Active workspaces based on Guarded attribute grammars with the notations introduced in Section 5 on a collaborative case management system. The real world scenario is observed from a disease surveillance system. Disease Surveillance as defined by the Centers for Disease Control and Prevention (CDC) [8] is the ongoing, systematic collection, analysis, interpretation, and dissemination of data about a health-related event for use in public health action to reduce morbidity and mortality and to improve wellbeing. It is a complex process that uses data from a plethora of sources and distributes its activities over several geographically dispersed actors with heterogeneous

profiles [6, 12, 44, 3]. Each actor plays a specific role in the system and offers a number of services needed by other stakeholders. Futhermore, the overall flow of tasks and activities is highly dependent on the available data and other contextual variables. For the purpose of the illustration, the scenario has been slightly adapted and does not therefore necessarily depict what happens in a real surveillance system.

The modeled scenario describes a situation in which three sets of actors with distinct roles (Epidemiologist, Physician, Biologist) actively participate in the surveillance and investigation of outbreaks of Influenza. An artifact in this scenario contains all the information pertaining to the treatment of a suspect case. The process starts with patient visits at a physician's office. The physician receives patients, registers the signs and symptoms, and verifies whether they correspond with those contained in the Influenza declaration criteria. If the verification is successful, he immediately declares the patient as a suspect case to the Disease Surveillance Center (DSC) (**caseDeclaration**). If the declared data contains saliva samples, the latter are sent to the biologist for laboratory analysis (**laboratoryAnalysis**). In parallel, the data is automatically analyzed by an epidemiologist (**dataAnalysis**) and eventually, outbreak alarms are produced. A number of verification tasks are run on the data and the analysis results to ascertain the alarm. The epidemiologist eventually creates a list of actions (*todo* list) that will have to be carried out by the physician to complete the alarm verification (**acmCheck**). Alongside these activities, he immediately informs Public Health Officials of the situation. Results from the laboratory analyses carried out by the biologist are used together with the results from the above checks to either confirm or revoke the outbreak alarm and produce an outbreak alert (**outbreakDecl**). Based on the outbreak characteristic data, the epidemiologist analyses the risks related to the outbreak alert and proposes appropriate counter measures.

The disease surveillance system consists of both the physicians and the Disease Surveillance Center (DSC). The latter proceeds to the analyses of the suspect cases transmitted by physicians.

```
diseaseSurveillance :: ⟨visit[physician], caseAnalysis⟩
where
physician = Alice | Bob | ...
visit :: % Role of a physician
caseAnalysis ::
    caseAnalysis (laboratoryAnalysis[biologist],
                  dataAnalysis[epidemiologist])
    where
    epidemiologist = Ann | Paul | ...
    biologist = Frank | Mary | ...
    caseAnalysis :: % Disease Surveillance Center
    laboratoryAnalysis :: % Role of a biologist
    dataAnalysis :: % Role of an epidemiologist
```

The case analysis is given by a hierarchical composition combining the roles of biologists and epidemiologists through an homonymous connector —the connector and the compound specification have the same axiom associated with service **caseAnalysis**.

The various components are modeled in detail in the following paragraphs. The following naming conventions are used throughout this section. Services (grammar axioms) are written in **bold**, sorts of internal (local) rules are unformatted, variable names are *italized*, and constructors are unformatted with first letter capitalized. Apart from Constructors, no other identifier has its first letter capitalized.

*Role of a Physician (*visit*).*

The physician receives patients, clinically examines them and if necessary, declares them as suspect cases of influenza.

```
visit(patient, alarm) =
 do (symps) ← clinicalAssessment(patient)
    () ← initialCare(symps)
    caseDeclaration(patient, symps, alarm)
```

First the physician fills out a form to report the symptoms observed in the patient. This information is a parameter of the rule (**input** clause) and thus is recorded in the artifact associated with the patient.

```
clinicalAssessment(patient) = input (symps)
```

Note that we use here the simplication rule $\mathbf{SR}_3$ meaning that the input value –the symptoms– is returned as a result of the clinical assessment. Then the physician can prescribe appropriate treatments.

```
initialCare(symps) = input (care)
                     return ()
```

Again the prescribed treatments are recorded in the artifact of the patient but this information is not returned as a result since it will not be used later.

If the symptoms correspond with the Influenza declaration criteria, the physician extracts some samples to be sent to a biologist for laboratory analysis and declares the patient as a suspect case. And he commits himself to later run some further verifications on the case –acmCheck– if required by the DSC, and the correspondinh results –checkRes– are send back to the DSC to complete the case analysis.

```
caseDeclaration(patient, symps) =
 input (samples)
 do (alarm) ← caseAnalysis (SuspectCase (patient,
                                         symps,
                                         samples),
                            checkRes)
    (checkRes) ← acmCheck(alarm)
    return ()
```

Note that this rule is not left-attributed: There are mutual dependencies between the subtasks **caseAnalysis** and acmCheck which are executed as coroutines: the caseAnalysis may produce an alarm that triggers the acmCheck, and the acmCheck returns check results which are used to continue the case analysis. If the patient does not have the influenza symptoms, the following rule is used instead.

```
caseDeclaration(patient, symps) = return ()
```

Finally if an alarm is raised –*alarm*=Alarm(*info*,*todo*)– containing analysis information together with a list of verification tasks, then the physician performs the corresponding

checks on the case and transmits the results.

$$\text{acmCheck}(\text{Alarm}(info, todo)) \;=\; \textbf{input}\ (checkRes)$$

Else, the following rule applies

$$\text{acmCheck}(\text{NoAlarm}) \;=\; \textbf{return}\ (\_)$$

The wildcard indicates a variable that is not instanciated –it is not defined by the rule– and whose value is not expected elsewhere.

Disease Surveillance Center.

If the patient is reported as a suspect case of influenza, a biologist and an epidemiologist are assigned to respectively carry out biological analyses for samples sent by the physician and to do statistical analyses and other disease surveillance related computations on the reported data in order to detect and investigate disease outbreaks. These analyses can produce an alarm.

$$\begin{array}{l}\textbf{caseAnalysis}\,(\text{SuspectCase}(patient, symps, samples),\\ \qquad\qquad checkRes) =\\ \quad \textbf{input}\ (bio, epi)\\ \quad \textbf{do}\ (labRes) \leftarrow \textbf{laboratoryAnalysis}[bio](samples)\\ \qquad \textbf{dataAnalysis}[epi](patient, symps, labRes, checkRes)\end{array}$$

Note the use of simplification rule $\mathbf{SR}_2$ due to the fact that the result returned by caseAnalysis is the result of **dataAnalysis**.

*Role of a Biologist (***laboratoryAnalysis***).*

The biologist receives the samples, verifies their conditioning and runs the requested analyses. The results are returned for used in confirming or revoking the outbreak alarm. For simplicity, we suppose and only model the case where the sample is well conditioned in which case the corresponding grammar is reduced to rule

$$\textbf{laboratoryAnalysis}(samples) \;=\; \textbf{input}\ (labResult)$$

*Role of an Epidemiologist (***dataAnalysis***).*

This service runs a number of automated data analyses and aggregation tasks on the entire declaration database with the aim of detecting aberrations from normal behaviour (outbreak alarms). This is followed by investigation tasks to better characterize the outbreak alarms. These investigative tasks (aspecific counter measure checks) may involve superposing the alarm data with information from other sources and with contextual variables. In this case study, we limit the investigative activities to a set of queries sent to the physician who declared the current case. This service is defined by the following rule,

$$\begin{array}{l}\textbf{dataAnalysis}(patient, symps, labResult, checkResult) =\\ \quad \textbf{do}\ (ack) \leftarrow \text{storeCaseData}(patient, symps)\\ \qquad \text{automatedAnalysis}(ack, labResult, checkResult)\end{array}$$

The epidemiologist stores the current case data in the surveillance database.

$$\text{storeCaseData}(patient, symps) \;=\; \textbf{return}\ (\text{Ack})$$

This triggers automated analyses on the entire database of declared suspect cases. The following rule is used if the analyses raise an alarm.

$$\begin{array}{l}\text{automatedAnalysis}(\text{Ack}, labResult, checkResult) =\\ \quad \textbf{input}\ (info, todo)\\ \quad \textbf{do}\ () \leftarrow \text{notifyAuth}(info)\\ \qquad () \leftarrow \text{outbreakDecl}(labResult, checkResult)\\ \qquad \textbf{return}\ (\text{Alarm}(info, todo))\end{array}$$

Note that the alarm is emitted in the first place, prior to notifying authorities and the outbreak declaration. And these two tasks can be performed concurrently. This clearly illustrates that the elements of a **do** body –including the **return** statement– are not necessarily executed in their order of appearance.

If on the other hand no alarm is raised the following rule is used instead.

$$\text{automatedAnalysis}(\text{Ack}, \_, \_) \;=\; \textbf{return}\ (\text{NoAlarm})$$

Observe that in both versions of rule automatedAnalysis pattern Ack is checked in order to ensure that the database has been updated according to the current case. This is an illustration of side effects as discussed in observation (O1), at the end of this section.

Raised alarms are immediately notified to public health authorities.

$$\text{notifyAuth}(info) \;=\; \textbf{return}\ ()$$

As earlier stated, the alarm contains a list of queries that will need to be answered by the physician who declared the suspect case. Rich with the investigation results and the laboratory results provided respectively by the physician and the biologist, the epidemiologist runs a number of alert analyses to confirm the outbreak alarm and declare an outbreak alert. He then analyses the risks involved and the public health impact of the outbreak. This information is used to propose appropriate counter measures which he communicates to public health authorities for action.

$$\begin{array}{l}\text{outbreakDecl}(labResult, checkResult) =\\ \quad \textbf{input}\ (alertInfos)\\ \quad \textbf{do}\ (risks) \leftarrow \text{riskAnalysis}(alertInfos)\\ \qquad (counterM) \leftarrow \text{defineCounterMeasures}(risks)\\ \qquad \text{feedback}(alertInfos, counterM)\end{array}$$

$$\text{riskAnalysis}(alertInfos) \;=\; \textbf{input}\ (risks)$$

$$\text{defineCounterMeasures}(risks) \;=\; \textbf{input}\ (counterM)$$

$$\begin{array}{l}\text{feedback}(alertInfos, counterM) \;=\\ \quad \textbf{input}\ (mail\_list)\\ \quad \text{sendFeedback}(mail\_list, alertInfos, counterM)\end{array}$$

$$\text{sendFeedback}(mail\_list, alertInfos, counterM) = \textbf{return}\ ()$$

If, conversely, the outbreak alert is not confirmed the following alternative rule is chosen.

$$\text{outbreakDecl}(labResult, checkResult) = \textbf{return}\ ()$$

Let us conclude this case study with some observations.

(O1) Some tasks may have side-effects. For instance sendFeedback forwards a message about the alert and the proposed counter measures to the email addresses given in *mail_list*. Also, storeCaseData stores the declared data in some local database on which the automated analysis is run. The automated analysis of the dabase produces information that guides the epidemiologist in her choice of raising an alarm or not (i.e., in choosing which rule to apply for automatedAnalysis). Such side-effects are not described in the model but they can easily be attached to rules when necessary.

(O2) Rules of the form

$$\text{task}(\mathit{args}) \;=\; \textbf{input}\ (\mathit{results})$$

corresponding to tasks that merely return values inputted by the user can be used for incremental specifications. Indeed if the rules specifying the resolution of task(*args*) are not yet designed, one can used the above temporary rule to obtain an approximate specification that can be executed and tested even though it will require the user to manually input the expected results. Then this temporary rule can progessively be refined to obtain a new rule where the results are no longer inputted by the user but synthesized from intermediate results produced by new subtasks that are introduced for that purpose.

(O3) The given specification is sound, which can be easily verified from the fact that the underlying grammar is non-recursive: A task never calls itself directly or indirectly. This situation, which is relatively common, provides a large family of sound specifications from which many other sound specifications can be built using hiearchical composition.

7. CONCLUSION

In this paper, we have proposed a declarative model of artifact-centric collaborative systems. The key idea was to represent the workspace of a stakeholder by a set of (mind)maps associated with the services that she delivers. Each map consists of the set of artifacts created by the invocations of the corresponding service. An artifact records all the information related to the treatment of the related service call. It contains open nodes corresponding to pending tasks that require user's attention. In this manner each user has a global view of the activities in which she is involved, including all relevant information needed for the treatment of the pending tasks.

Using a variant of attribute grammars, called guarded attribute grammars, one can automate the flow of information in collaborative activities with business rules that put emphasis on user's decisions. We gave an in-depth description of this model through its syntax and its behaviour. We paid attention to two crucial properties of this model. First, the input-enabled GAG satisfy a monotony property that allows to distribute the model on an asynchronous architecture. Input-enabledness is undecidable but we have identified the sufficient condition of strongly-acyclicity that can be checked very efficiently by a fixpoint computation of an over-approximation of attribute dependencies. Second, soundness is a property that asserts that any case introduced in the system can reach completion. This property is also undecidable but we have defined a hierarchical composition of GAGs that preserves soundness and thus allows to build large specifications that are sound by construction if one starts from small components which are known to be sound.

We have introduced some notations and constructs paving the way towards an expressive and user-friendly specification language for active workspaces Also, we have demonstrated the expressive power and exemplified the key concepts of active workspaces on a case study for a disease surveillance system

In the rest of this section we list key features of the model and draw some future research directions.

7.1 Assessment of the Model

In a nutshell *active workspaces and guarded attribute grammars provide a modular, declarative, user-centric, data-driven, distributed and reconfigurable model of case management.* It favors flexible design and execution of business process since it possesses (to varying degrees) all four forms of *Process Flexibility* proposed in [40].

Concurrency.

The lifecycle of a business artifact is implicitly represented by the grammar productions. A production decomposes a task into new subtasks and specifies constraints between their attributes in the form of the so-called semantic rules. The subtasks may then evolve independently as long as the semantic rules are satisfied. The order of execution, which may depend on value that are computed during process execution, need not (and cannot in general) be determined statically. For that reason, this model dynamically allows maximal concurrency. In comparison, models in which the lifecycle of artifacts are represented by finite automata constrain concurrency among tasks in an artificial way.

Modularity.

The GAG approach also facilitates a modular description of business processes. For instance, when laboratory test requests are sent to the biologist, the physician needs not know about the subprocess through which the specimen will pass before results are finally produced. For instance, the service *laboratoryAnalysis* can –in a more refined specification– be modeled by a large set of rules including specimen purification, and several biological and computational processes. However, following a top-down approach, one simply introduces an attribute in which the results should eventually be synthesized and delegate the actual production of the expected outcome to an additional set of rules. The identification of the different roles involved in the business process can also contributes to enhance modularity. Finally, some techniques borrowed from attribute grammars, like descriptional composition [20, 21], decomposition by aspects [42, 41] or higher-order attribute grammars [43, 38], may also contribute to better modular designs.

Reconfiguration.

The workflow can be reconfigured at run time: New business rules (associated with productions of the grammar) can be added to the system without disturbing the current cases. By contrast, run time reconfiguration of workflows modeled by Petri nets (or similar models) is known to be a complex issue [13, 17]. One can also add "macro rules" corresponding to specific compositions of rules. For instance if the Editor-in-chief wants to handle the evaluation of a paper, he can decide to act as an associate editor and as a referee for this particular submission. However, this means forwarding the corresponding case to himself as an associate editor and then asking himself as a referee if she is willing to write a report. A more direct way to model this decision is to encapsulate these steps in a compound macro production that bypasses the intermediate communications. More generally compound rules can be introduced for handling unusual behaviors that deviates from the nominal workflow.

Logged information.

When a case is terminated, the corresponding artifact collects all relevant information of its history. Nodes are labeled by instances of the productions that have lead to the completion of the case. Henthforth, they record the decisions (the choices among the allowed productions) together with information associated with these decisions. In the case of the editorial process, a terminated case contains the names of the referees, the evaluation reports, the editorial decision, etc. A terminated case is a tree whose branches reflect causal dependencies among subactivities used to solve a case, while abstracting from concurrent subactivities. The artifacts can be collected in a log which may be used for the purpose of process mining [39] either for process discovery (by inferring a GAG from a set of artifacts using common patterns in their tree structure) or for conformance checking (by inspection of the logs produced during simulations of a model or the executions of an actual implementation).

Distribution.

Guarded attributed grammars can easily be implemented on a distributed architecture without complex communication mechanisms –like shared memory or FIFO channels. Stakeholders in a business process own open nodes, and communicate asynchronously with other stakeholders via messages. Moreover there are no edition conflicts since each part of an artifact is edited by the unique owner of the corresponding node. Morevover, the temporary information stored by the attributes attached to open nodes no longer exist when the case has reached completion. Closing nodes eliminates temporary information without resorting to any complex mechanism of distributed garbage collection.

7.2 Future Works

An immediate milestone is the design of a prototype of the Active Workspaces runtime environment. This prototype shall contain support tools (editor, parser, checker, simulators ...) to analyze, implement, and run GAG descriptions of AW systems. The following research directions are also considered:

Coupling GAG systems with external features.

In Section 6, we have imagined a scenario where the reported cases are stored in some local database and the analysis and investigation tasks directly access and use these data. Such implicit side-effects of rules abound –see Observation (O1) Section 6. They generally do not conflict with the GAG specification but rather complement it, in basically two ways. First, they allow to associate real-world activities with a rule, like extracting samples from a patient –caseDeclaration–, sending messages –sendFeedback–, performing verifications –acmCheck, riskAnalysis–, etc. Second, they may be used to extract information from the current artifacts to build dashboards or to feed some local database that are later used to guide the user on her choice of the rule to apply for a pending task. They may, in a more coercitive fashion, suggest a specific rule to apply or even inhibit some of the rules. Some information from dashboards or contained in a local database can also be used to populate some input parameters of a rule in place of the stakeholder. The actions of the stakeholder, namely choosing which rule to apply and the values to input in the system are left unspecified in the GAG specification –they constitute its only form on non-determinism. Side-effects can thus complement the GAG specification by providing an additional support to the stakeholder in this regard. Nonetheless, if we resort to a distant database or web services then it will be necessary to put some restrictions on the allowed queries to preserve monotony and thus to guarantee a safe distribution of the specification. Similarly we must be careful, if side-effects can inhibit some rules, that this does not jeopardize soundness. By the way, it is important to dispose of a language to describe side-effects of rules and in particular a language for making queries on active workspaces. The ideal solution would be to implement GAG as a domain specific language embedded into a general purpose language. In that case we could directly write the side-effects in the host language.

Development methodology.

We need to develop a support for the derivation of a GAG specification from a problem description. Object-oriented programming uses, for that purpose, normalized notations and diagrams for specifying the involved classes, uses cases, activities and collaborations. A modeling language for GAG should concentrate on the central concepts of the model: The artifacts, task decomposition, user's decisions, user's communication. For the latter one may use concepts of speech act theory [4, 45] for classifying business rules in terms of assertions, orders, requests, commitments, etc. As far as artifacts are concerned, one can observe on the two examples of the paper (Editorial process and Disease surveillance) that we have very few completed artifacts, once the case's specific information contained in the artifacts have been abstracted, One can try to extract the business rules –task decomposition and semantic rules– starting from these archetypal artifacts and answering the following questions: What are the dependencies between data field values? Who produces these values? What information does one need to

produce that value? Can one identify the conditions that justify variabilities between similar artifacts? etc. As a formalism for distributed collaborative systems the GAG model should also come with a complete method for elaborating the procedure going from a problem description, through the implementation, to the deployment on a distributed asynchronous architecture. Just as with Software Processes [37], this method will provide notations which describe how to identify relevant information (roles, data, processes ...) and propose appropriate representation tools to add expressiveness to the textual descriptions of collaborative case management systems.

Applicability and Pertinence.

The development of case studies is very important to check the pertinence, level of applicability, and practical limitations of the GAG model. These case studies are also important to refined the specification language that we have sketched in Section 5 and to extract from practise useful concepts for a modeling language. We continue our study on Disease surveillance that we intend to effectively implement on a real situation in collaboration with epidemiologists from Centre Pasteur in Cameroon. Besides Disease surveillance, it is also useful to develop more representative case studies of distributed collaborative systems. The following two examples are representative case studies in which our model can provide advantages over existing techniques. **Reporting systems** where several stakeholders collaborate to build a report. The grammar can reflect the structure of the report, the identification of stakeholders and their respective contributions. The semantic rules implement the automatic assembly of the report from the bits provided by the distributed stakeholders. Most often, many information to be inserted in a report are already available. Semantics rules avoid redundancies and reduce workload: You *write only once* each piece of information, it is then collected in a synthesized attribute for further use. Guarded attribute grammars also avoids *email overload* –a problem that appears frequently when you have to coordinate a group of people to complete a task– since most of the communication is directly made between the active workspaces. A **distributed distance learning** which is a highly decentralized system deployed mostly in degraded environments (where Internet connection is not always available) with most stakeholders working off-line and synchronizing their activities upon establishment of an Internet connection. Also, the declarative decomposition of learning activities, which do not impose particular execution order, gives more flexibility in the design and description or learning activities and more freedom to the learner in the learning path.

8. REFERENCES

[1] S. Abiteboul, J. Baumgarten, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing distributed workspaces with active XML. In *VLDB*, pages 1061–1064, 2003.

[2] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *BDA'02*, 2002.

[3] P. Astagneau and T. Ancelle. *Surveillance épidémiologique*. Lavoisier, 2011.

[4] J.L. Austin. *How to Do Things with Words*. Oxford Univ. Press, 1962.

[5] K. Backhouse. A functional semantics of attribute grammars. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 2280 of *LNCS*, pages 142–157. Springer, 2002.

[6] H. Chen, D. Zeng, and P. Yan. *Infectious Disease Informatics: Syndromic Surveillance for Public Health and Bio-Defense*. Springer, 1st edition, 2009.

[7] L.M. Chirica and D.F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.

[8] World Health Organization / Centers For Disease Control. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. Technical report, WHO/CDC, Georgia, USA, 2001.

[9] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes i and ii. *Theor. Comput. Sci.*, 17:163–191 and 235–257, 1982.

[10] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst.*, 37(3):22, 2012.

[11] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. *Inf. Syst.*, 38(4):561–584, 2013.

[12] V. Dato, R. Shephard, and M.M. Wagner. Chapter 2 - outbreaks and investigations. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 13 – 26. Academic Press, Burlington, 2006.

[13] G. De Michelis and C.A. A. Ellis. Computer supported cooperative work and Petri nets. In *Advanced Course on Petri Nets, Dagstuhl 1996*, volume 1492 of *LNCS*, pages 125–153. Springer, 1998.

[14] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *J. Log. Program.*, 2(2):119–155, 1985.

[15] P. Deransart and J. Maluszynski. *A grammatical view of logic programming*. MIT Press, 1993.

[16] S. Doaitse Swierstra, P.R. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[17] C.A. Ellis and K. Keddara. Ml-dews: Modeling language to support dynamic evolution within workflow systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000.

[18] R. Eshuis, R. Hull, Y. Sun, and R. Vaculín. Splitting GSM schemas: A framework for outsourcing of declarative artifact systems. In *BPM*, volume 8094 of *LNCS*, pages 259–274. Springer, 2013.

[19] J. Fokker. Functional parsers. In Springer-Verlag, editor, *Advanced Functional Programming, First International Spring School*, volume 925 of *LNCS*, pages 1–23, 1995.

[20] H. Ganzinger. Increasing modularity and language-independency in automatically generated

compilers. *Sci. Comput. Program.*, 3(3):223–278, 1983.

[21] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *SIGPLAN Symposium on Compiler Construction*, pages 157–170. ACM, 1984.

[22] Object Management Group. Business process model and notation, v 2.0. Technical report, OMG, 2011. http://www.bpmn.org/.

[23] L. Hélouët and A. Benveniste. Document based modeling of web services choreographies using active XML. In *ICWS*, pages 291–298. IEEE Computer Society, 2010.

[24] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM 2008*, volume 5332 of *LNCS*, pages 1152–1163. Springer, 2008.

[25] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F.T. Heath, S. Hobson, M.H. Linehan, S. Maradugu, A. Nigam, P.N. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011*, pages 51–62. ACM, 2011.

[26] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, FPCA*, volume 274 of *LNCS*, pages 154–173. Springer, 1987.

[27] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transaction on Software Engineering Methodologies*, 14(3):331–380, 2005.

[28] D.E. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2):127–145, 1968.

[29] N. Lohmann and K. Wolf. Artifact-centric choreographies. In *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010.*, pages 32–46, 2010.

[30] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[31] B.H. Mayoh. Attribute grammars and mathematical semantics. *SIAM J. Comput.*, 10(3):503–518, 1981.

[32] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42:428–445, July 2003.

[33] OASIS. Web services business process execution language. Technical report, OASIS, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[34] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[35] J. Saraiva and S.D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In *Generative Programming and Component Engineering, GPCE 2003*, volume 2830 of *LNCS*, pages 307–323. Springer, 2003.

[36] J. Saraiva, S.D. Swierstra, and M.F. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction, CC 2000*, volume 1781 of *LNCS*, pages 279–294. Springer, 2000.

[37] I. Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2011.

[38] S.D. Swierstra and H. Vogt. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 256–296. Springer, 1991.

[39] W.M.P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[40] W.M.P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.

[41] E. Van Wyk. Implementing aspect-oriented programming constructs as modular language extensions. *Sci. Comput. Program.*, 68(1):38–61, 2007.

[42] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Compiler Construction, ETAPS 2002, Grenoble, France*, pages 128–142, 2002.

[43] H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

[44] M.M. Wagner, L.S. Gresham, and V. Dato. Chapter 3 - case detection, outbreak detection, and outbreak characterization. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 27 – 50. Academic Press, Burlington, 2006.

[45] T. Winograd and F. Flores. *Understanding Computer and Cognition: A new Foundation for Design*. Norwood, 1986.